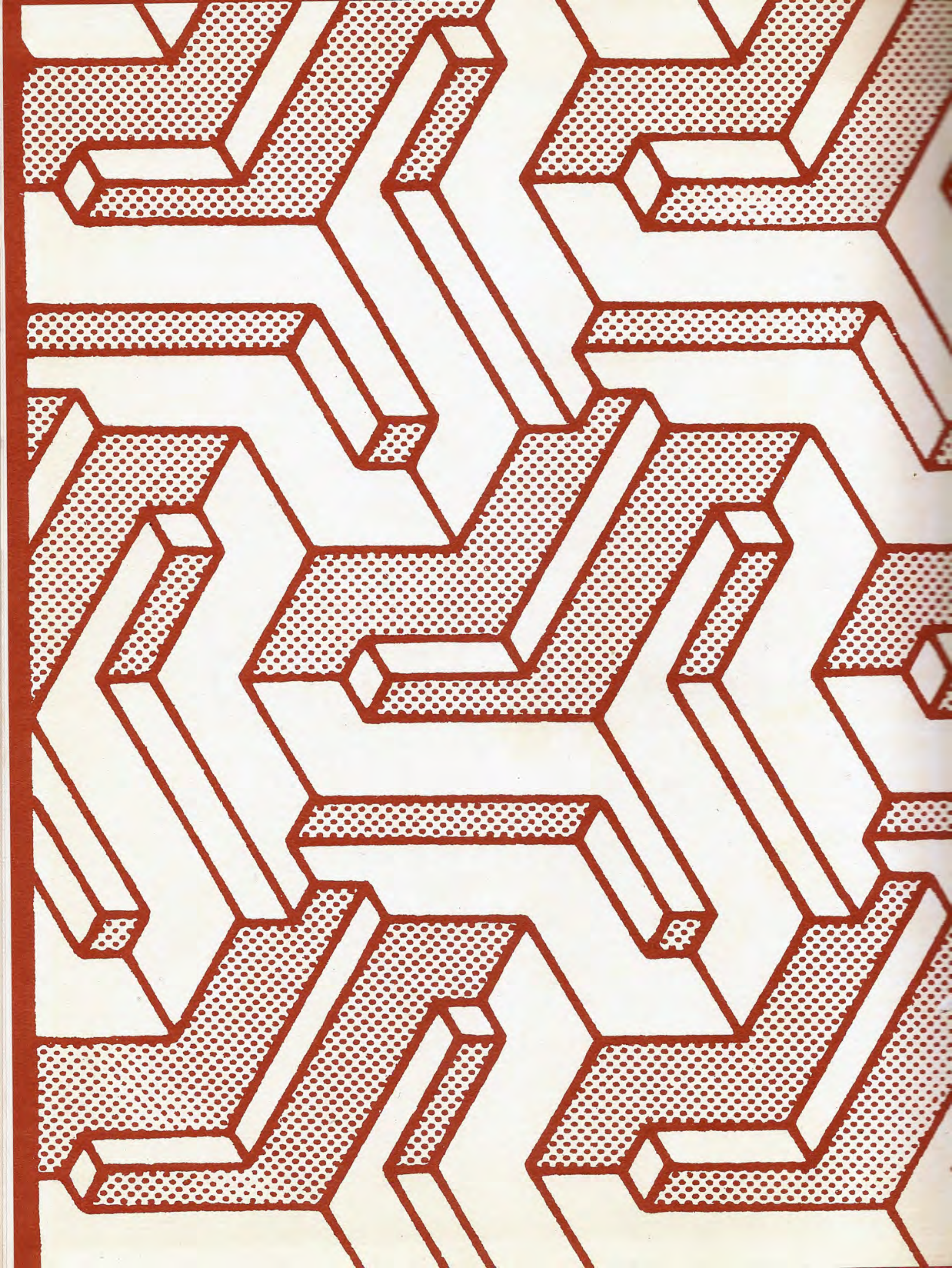
GRAN ENCICLOPEDIA INFORMATICA

LENGUAJES / 1

EDICIONES NUEVA LENTE



GRAN ENCICLOPEDIA INFORMATICA

EDICIONES NUEVA LENTE


AMSTRAD 256K ORD. PERSONAL PROC. DE TEXTOS

SUMARIO

Lenguajes informáticos	5	Del código máquina a los lenguajes de alto nivel
ADA	11	Un paso importante hacia la unificación
C (1)	19	Un lenguaje para la programación de sistemas
C (y 2)	35	Estructuras de datos: arrays y registros
COBOL	47	Un pionero de los lenguajes informáticos
FORTH (1)	57	Introducción al lenguaje FORTH
FORTH (2)	69	Palabras y estructuras de control
FORTH (3)	81	Entrada de datos y bases de numeración
FORTH (y 4)	93	Números de doble precisión, arrays y últimos detalles
FORTRAN	105	El precursor de los lenguajes de alto nivel
LISP	115	Creando programas inteligentes

Una publicación:

Ediciones Nueva Lente, S. A.

Director editor: MIGUEL J. GOÑI

Director de producción: SANTOS ROBLES.

Director de la obra: FRANCISCO LARA.

Colaboradores: PL/3 - MANUEL MUÑOZ - ANGEL MARTINEZ - MIGUEL DE ROSENDO - DAVID SANTAOLALLA - SANTIAGO RUIZ - LUIS COCA - MIGUEL ANGEL VILA - MIGUEL ANGEL SANCHEZ VICENTE ROBLES.

Diseño: BRAVO/LOFISH.

Maquetación: JUAN JOSE DIAZ SANCHEZ.

Ilustración: JOSE OCHOA.

Fotografía: (Equipo Gálata) ALBINO LOPEZ y EDUARDO AGUDELO.

Ediciones Nueva Lente, S. A.:
Dirección y Administración:
Benito Castro, 12. 28028 Madrid. Tel.: 245 45 98.

Números atrasados y suscripciones:
Ediciones Ingelek, S. A.
Plaza de la Rep. Ecuador, 2 - 1.°. 28016 Madrid.
Tel.: 250 58 20.

Plan general de la obra:
18 tomos monográficos de aparición quincenal.

Distribución en España:
COEDIS, S. A. Valencia, 245. Tel.: 215 70 97.
08007 Barcelona.

Delegación en Madrid:
Serrano, 165. Tel.: 411 11 48.

Distribución en Argentina:
Capital: AYERBE
Interior: DGP

Distribución en Chile: Alfa Ltda.

Distribución en México:
INTERMEX, S. A.
Lucio Blanco, 435
México D.F.

Distribución en Uruguay:
Ledian, S. A.

Edita para Chile:
PYESA
Doctor Barros Borgoño, 123
Santiago de Chile

Importador exclusivo Cono Sur:
CADE, SRL. Pasaje Sud América, 1532.
Tel.: 21 24 64. Buenos Aires - 1.290. Argentina.

Fotomecánica: Ochoa, S. A.
Miguel Yuste, 32. 28037 Madrid.

Impresión: Gráficas Reunidas, S. A.
Avda. de Aragón, 56. 28027 Madrid.

ISBN de la obra: 84-7534-184-5.
ISBN del tomo 11: 84-7534-231-0

Printed in Spain
Depósito legal: M. 27.605-1986

Precio de venta al público en Canarias, Ceuta y Melilla: 940 ptas.
Febrero 1987

Lenguajes informáticos

Del código máquina a los lenguajes de alto nivel



Ocurre en el universo de los seres humanos y, por supuesto, también debía suceder algo semejante en el mundo de los ordenadores. Las máquinas programables creadas por el hombre forman una torre de Babel en la que la disparidad de los lenguajes de comunicación es tan acusada como pueda serlo en la sociedad humana. Al igual que en el terreno de los lenguajes humanos hay algunos idiomas con mayor relevancia y difusión, utilizados como patrón de entendimiento (el inglés, el español o el francés), en la Babel de los ordenadores también existen algunos lenguajes con acusado protagonismo. Sin lugar a dudas el más popular de ellos es el BASIC, aunque poco a poco va cediendo ante los lenguajes estructurados (como el Pascal), que unen a su potencia una facilidad de manejo y comprensión mucho mayor.

No terminan aquí las analogías. En la sociedad humana, muchos idiomas presentan dialectos o variantes, más o menos distantes de la raíz original. Una realidad trasladable a los lenguajes informáticos, donde los dialectos y versiones del BASIC son casi tan numerosos como los modelos de ordenadores. Esta situación habrá quedado particularmente clara para los que hayan al menos hojeado los tomos dedicados a tan popular lenguaje en esta enciclopedia.

Cualquier exposición relativa a los lenguajes informáticos debe partir del personaje central: el ordenador. Sabemos que es una máquina cuya propiedad diferenciadora reside en que admite una programación que le permitirá realizar una ingente variedad de tareas; tantas como el usuario sea capaz de programar.

Debido a su constitución interna —un conjunto de circuitos electrónicos, de tecnología digital—, el ordenador está capacitado para entender un lenguaje construido a partir de unidades elementales de información, o bits (un cero o un uno, representados, normalmente, por la presencia o ausencia de tensión o corriente, que un circuito electrónico es capaz de detectar). A partir de este alfabeto mínimo (0 y 1) pueden construirse palabras (dependiendo de la unidad central de proceso, una palabra puede estar formada por ocho, dieciséis, treinta y dos o más ceros y unos, siempre en potencias de dos) y, por supuesto, frases (que no serían otra cosa que un conjunto de palabras puestas una detrás de otra, sin elementos separadores: no olvidemos que un microprocesador no entiende de espacios) con las que establecer una comunicación repleta de contenido.

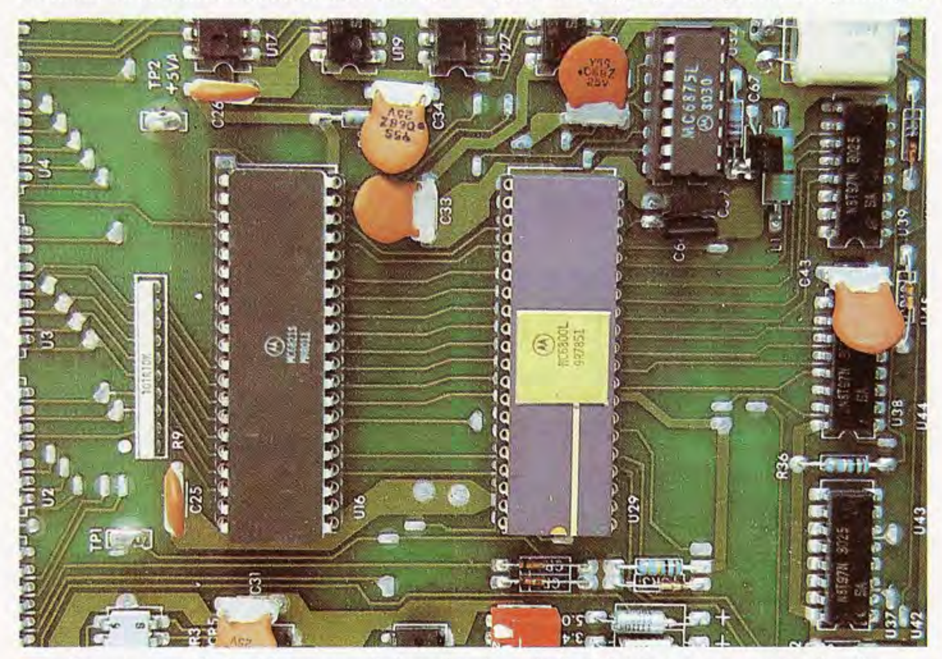
Algo es ya evidente: por su naturaleza, el ordenador está capacitado para dialogar en un lenguaje particular, que por sus características denominaremos lenguaje máquina, edificado a partir de las unidades mínimas de información, ceros y unos. Estos elementos, los BITs (contracción de su denominación inglesa, Binary digIT: dígito binario), coinciden con los propios del sistema de numeración binario.

Niveles de los lenguajes informáticos

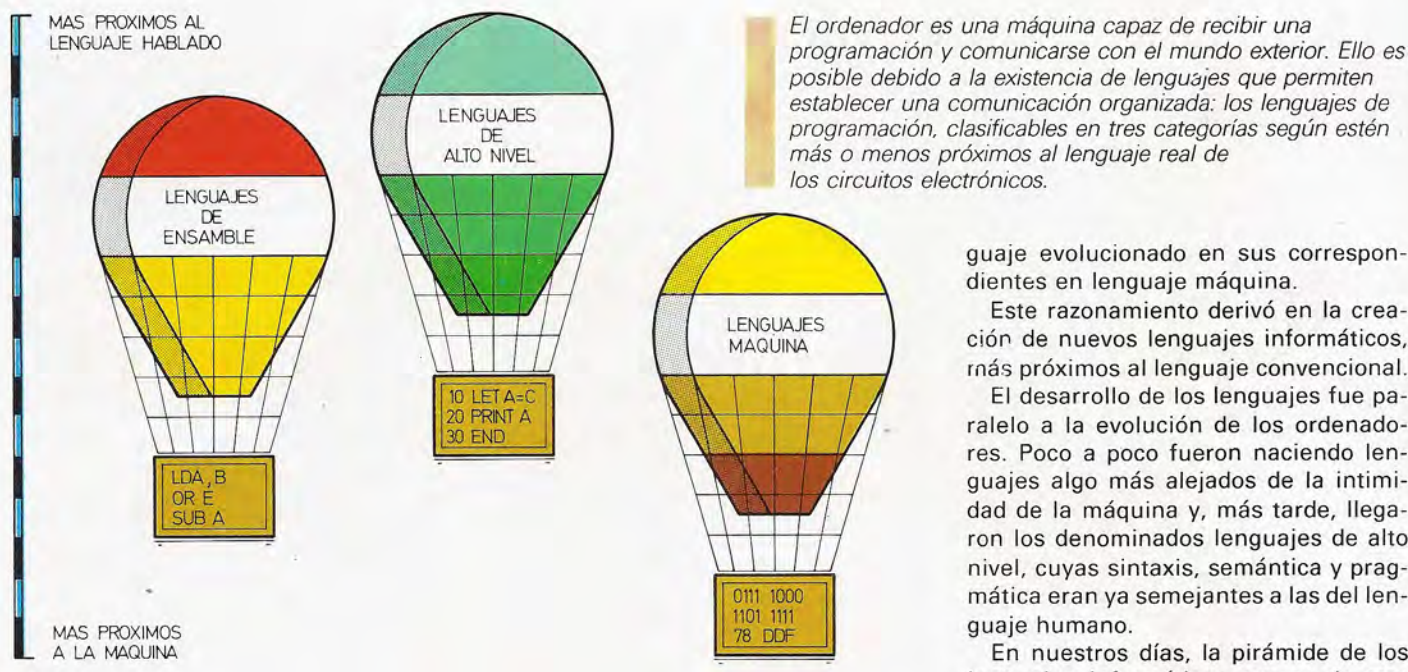
La capacidad del ordenador para intercambiar información con el usuario es un hecho ya precisado. Por el momento, hemos visto que es posible establecer una comunicación con sus circuitos electrónicos utilizando su lenguaje «materno»: el código máquina. Una simple reflexión acerca de esta vía de diálogo nos lleva a algunas conclusiones no excesivamente favorables para el lenguaje máquina. En principio,



La disparidad de lenguajes y la necesidad de medios de traducción que permitan el entendimiento son realidades compartidas por los lenguajes humanos convencionales y los lenguajes de programación.



En su intimidad, el lenguaje de los circuitos de la máquina es una complicada secuencia de niveles de tensión: estados de conexión y desconexión que representamos por medio de combinaciones de unos y ceros.



El ordenador es una máquina capaz de recibir una programación y comunicarse con el mundo exterior. Ello es posible debido a la existencia de lenguajes que permiten establecer una comunicación organizada: los lenguajes de programación, clasificables en tres categorías según estén más o menos próximos al lenguaje real de los circuitos electrónicos.

no cabe duda de que se trata de un lenguaje difícil de aprender para el usuario; no sólo por la complejidad inherente a su estructura, sino también por el hecho de que su íntima relación con la máquina obliga a conocer muy a fondo los entresijos de ésta. Además, hay que recordar que el ordenador es una máquina ignorante, a la que hay que instruir con toda suerte de detalles mínimos; una tarea ardua y difícil de llevar a buen término a base de combinar unos y ceros.

Aunque la programación en lenguaje de máquina no está tan lejana en el tiempo (ciertamente, nada relacionado con los ordenadores puede ser demasiado antiguo), pronto los diseñadores/operadores de ordenadores (en sus comienzos, sólo el equipo que había creado una máquina era capaz de entenderla y programarla) empezaron a cuestionarse la necesidad de su empleo: al fin y al cabo, el ordenador no es más que una herramienta creada por el hombre para facilitar su trabajo... ¿Por qué hay que acondicionarse a la máquina, si cabe la posibilidad de comunicarse con ella utilizando un lenguaje próximo al humano?. Todo el problema se reduciría a crear los traductores adecuados para convertir las descripciones formuladas en lenguaje evolucionado en sus correspondientes en lenguaje máquina.

En los lenguajes de máquina simbólicos, el programador utiliza códigos nemotécnicos en lugar de códigos numéricos. La programación es enormemente más sencilla que empleando únicamente unos y ceros, aunque la tarea del usuario continúa siendo enorme.

Este razonamiento derivó en la creación de nuevos lenguajes informáticos, más próximos al lenguaje convencional.

El desarrollo de los lenguajes fue paralelo a la evolución de los ordenadores. Poco a poco fueron naciendo lenguajes algo más alejados de la intimidad de la máquina y, más tarde, llegaron los denominados lenguajes de alto nivel, cuyas sintaxis, semántica y pragmática eran ya semejantes a las del lenguaje humano.

En nuestros días, la pirámide de los lenguajes informáticos consta de tres niveles:

Lenguajes máquina

Ocupan el estrato inferior, menos evolucionado, de los lenguajes informáticos. Dada su total consonancia con la naturaleza íntima de la máquina, es obvio que el lenguaje será distinto según se trate de unos u otros ordenadores.

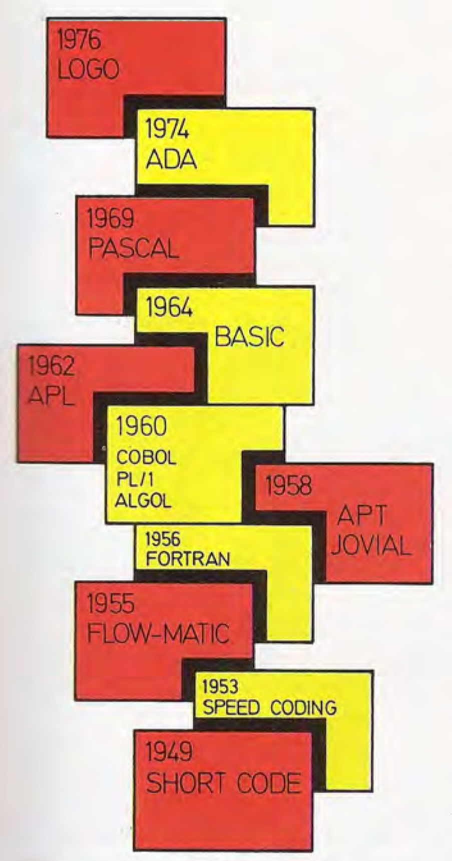
Cabe recordar que el cerebro o unidad central de proceso de un ordenador es el microprocesador; en consecuencia, el lenguaje máquina apropiado para cada ordenador dependerá del tipo de microprocesador que resida en su núcleo. En efecto, cada microprocesador (6800, 6502, Z80, ó, más recientemente, 8088, 8086, 68000, etc.) tiene su propio lenguaje máquina.

Lenguajes ensambladores

El estrato intermedio está ocupado por los lenguajes de ensamblado, o ensambladores. El repertorio de elementos que intervienen en la confección de programas coincide, en este caso, con conjuntos de símbolos o nemónicos, que ofrecen una mayor comodidad que las asociaciones de ceros y unos.

Su relación con el lenguaje máquina es muy próxima, hasta el punto de que cada familia de microprocesadores posee un lenguaje ensamblador propio, en directa correspondencia con su lenguaje máquina.

La tarea de confección y corrección de los programas resulta ahora más fácil,



A medida que ha evolucionado la informática han ido creándose nuevos lenguajes de programación «de alto nivel», con una estructura semejante a la de los lenguajes humanos.

dada la comodidad que supone el emplear grupos de letras en lugar de ceros y unos para definir las operaciones. No cabe duda de que para incrementar un número en una unidad es más grato escribir INC A que 00111100 01000001; y, desde luego, la posibilidad de cometer un error es bastante menor.

El lenguaje ensamblador tiene, a pesar de su dificultad, un ámbito de aplicación del que ningún lenguaje de alto nivel ha sido capaz de desplazarlo, aunque se haya intentado, y se siga haciéndolo, con verdadero interés: el desarrollo de aplicaciones en las cuales la velocidad de proceso sea crítica. Un programa creado directamente en ensamblador puede ser optimizado al máximo, algo que un compilador, que actúa según reglas generales de traducción, no puede hacer. Entre estas aplicaciones podemos citar el control de procesos en tiempo real, y el desarrollo de sistemas operativos (el sistema operativo es un programa residente en el ordenador que le indica cómo atender a los distintos elementos a él asociados: teclado, pantalla, unidad de disco, etc. Sin sistema operativo el ordenador se convierte en una máquina totalmente inútil). Poco a poco, sin embargo, el ensamblador está siendo desplazado incluso de estas tareas críticas. Por poner un ejemplo, en el sistema operativo UNIX, uno de los más recientes, únicamente el núcleo está programado en ensamblador. El resto se ha desarrollado en el lenguaje de alto nivel C.

Lenguajes de alto nivel

Estos son ya lenguajes evolucionados que mantienen un gran paralelismo con los lenguajes hablados convencionales. En este tercer nivel, la disparidad de los lenguajes no es achacable al procesador, sino al fin a que han sido destinados: un lenguaje de gestión de procesos es enormemente distinto de otro pensado para la realización de cálculos matemáticos, y uno de propósito general, como pueda serlo el BASIC, es totalmente diferente de ambos. Los lenguajes de alto nivel más difundidos (BASIC, Pascal, FORTRAN, COBOL, LOGO, ...) disponen de traductores para su conversión al lenguaje máquina de casi cualquier microprocesador. Como ya sabemos, estos traductores pueden ser intérpretes (traducen y ejecutan las sentencias una a una) o compiladores (se traduce el programa completo antes de proceder a su ejecución).

Las ventajas son evidentes: la redacción del programa resulta comprensible para cualquier usuario, y, por tanto, es más cómoda su redacción y la detección de posibles errores sintácticos. Por lo demás se reduce el tiempo de programación y, lo que es más importante, cabe ya pensar en que un mismo programa puede ser ejecutado por distintos ordenadores.

Lenguajes de alto nivel

El BASIC es, sin lugar a dudas, el lenguaje de alto nivel más popular en nuestros días. Un lenguaje de programación polivalente, que ha derivado en múltiples dialectos: desde versiones resumidas para el aprendizaje, hasta potentes y evolucionadas versiones orientadas a la programación profesional. Pero no arranca de ahí la historia de los lenguajes informáticos. Antes del BASIC y después de él son muchos de los lenguajes que han visto la luz: desde el SHORT CODE que en 1949 desarrolló el doctor Mendy para la firma UNIVAC, o el SPEED CODING nacido en 1953 con destino a IBM, hasta los más recientes como el LOGO (1976) o el ADA (1980), un lenguaje específicamente diseñado para tenerlo como estándar en el Departamento de Defensa de Estados Unidos, lo que le asegura ya desde el principio una amplia difusión.

La clasificación de los lenguajes de alto nivel es una misión casi imposible, debido a que cada día aparecen nuevos lenguajes o dialectos de los ya existentes. En el año 1980 estaban registrados unos doscientos lenguajes diferentes, muchos de los cuales estaban especializados en la resolución de un determinado tipo de problemas o sólo eran uti-



El ordenador puede encargarse de traducir los programas escritos en un lenguaje de alto nivel a código máquina. Para ello es necesario ejecutar un programa «traductor», que utilice las instrucciones del programa fuente.

lizables por un determinado tipo de ordenadores.

Otra dificultad adicional está en poder determinar a qué categoría pertenece un lenguaje concreto. De una forma muy general podemos elaborar la clasificación que sigue, si bien los diferentes grupos no son en absoluto disjuntos.

a) Lenguajes científicos.

Históricamente son los primeros lenguajes evolucionados, debido a dos factores primordiales: en principio, la formulación matemática permite una más fácil formalización del lenguaje y, en segundo lugar, muchas de las aplicaciones científicas tienen un carácter poco repetitivo, por lo que resulta muy importante reducir el tiempo de programación.

Los primeros lenguajes de este tipo fueron el SHORT CODE, creado en 1949 por el doctor Mandy para la casa UNIVAC, y el SPEED CODING, desarrollado en 1953 por Backus y Seldon para IBM. Tras ellos, y antes de llegar al más usado, el FORTRAN, aparecieron MATH-MATIC, UNICODE, IT, GAT Y FORTRANSIT, entre otros.

En la actualidad, los lenguajes más conocidos son ALGOL, FORTRAN en sus diferentes versiones, APL, BASIC y Pascal. Aunque su origen sea el desarrollo de aplicaciones científicas, muchos de ellos se utilizan normalmente como lenguajes de propósito general.

b) Lenguajes de gestión.

Los lenguajes de gestión están orientados a la solución de problemas de tratamiento de datos para la gestión, por lo que predominan en ellos las instrucciones especializadas en el tratamiento de procesos de entrada/salida.

El primer lenguaje creado para aplicaciones en este campo fue el FLOW-MATIC, desarrollado en 1955 por el doctor Hopper para UNIVAC. En la actualidad, el más conocido y utilizado es el COBOL.

c) Lenguajes polivalentes.

Son el resultado del intento de obtener un lenguaje que cubriera tanto el área científica como el área de gestión, de una forma equilibrada.

El primero que cumplió estos objetivos fue JOVIAL, desarrollado en 1959 por el Strategic Air Command Control System. Tras él surgió en 1964 el PL/1.

Otros lenguajes de esta categoría son FORTH, ALGOL, ADA Y LOGO, este último de gran importancia en el campo de la educación.

d) Lenguajes para procesado de listas y cadenas.

Es éste un grupo muy especializado, del que forman parte el IPL-V, el LISP, el COMIT y el SNOBOL.

e) Lenguajes para expresiones algebraicas formales.

Este tipo de lenguajes permiten el uso de expresiones matemáticas sin referirse a valores numéricos concretos y, por tanto, no necesitan un fuerte desarrollo de Análisis Numérico.

El primero fue el ALGY, creado en 1961, aunque también cabe citar el FORMAC, MATHLAB, ALTRAN, FLAP, MAGIC PAPER y SML.

f) Lenguajes de manejo de ficheros y bancos de datos.

El incremento de la cantidad de información que hay que manipular dentro de un proceso obligó a perfeccionar la gestión de los datos. Ante esta necesidad se empezaron a desarrollar lenguajes y sistemas para el tratamiento de ficheros y bases de datos. Estos sistemas suelen integrarse bajo las siglas IMS (Information Manegement System, o Sistema de Manejo de Información), DMS (Data Management System, o Sistema de Manejo de Datos), DBS (Data Base System, o Sistema de Base de Datos), etc.

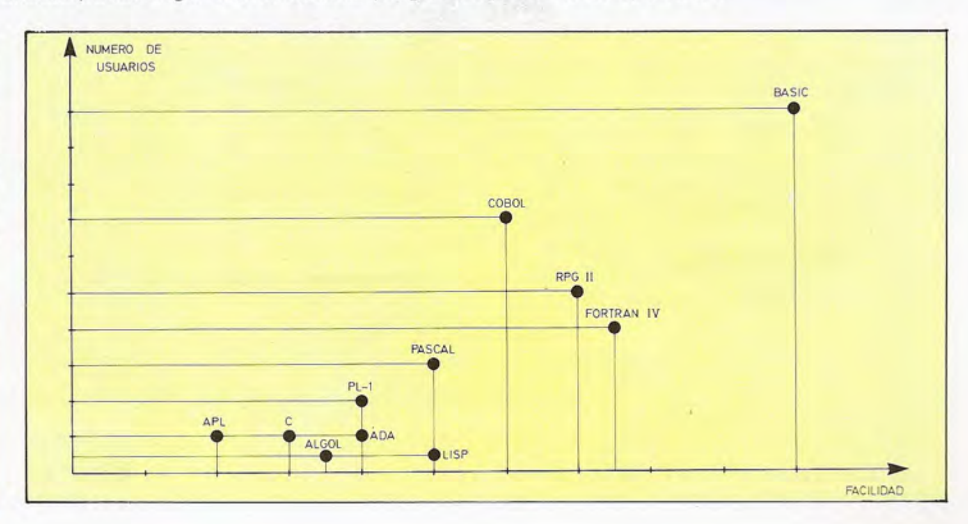
g) Lenguajes para la inteligencia artificial.

Este campo, de gran desarrollo en los últimos tiempos, precisa de lenguajes

Dentro del universo de los lenguajes de programación caben desde lenguajes orientados al diálogo especializado (gestión, cálculo, enseñanza,...), hasta lenguajes polivalentes útiles para programar cualquier tipo de aplicación.

Con los lenguajes de alto nivel la programación de los ordenadores no exige un profundo conocimiento de su estructura interna, con lo que cualquier usuario no especializado puede llegar a confeccionar programas plenamente operativos.

de enorme capacidad de razonamiento lógico y manejo de listas y palabras, en los que el cálculo queda relegado a un segundo plano. Entre ellos cabe destacar como más importante a LISP (LISt Processing, o procesado de listas) y PROLOG (PROgramming in LOGic, o programación lógica), con sus diversos



A pesar de la gran cantidad de lenguajes de alto nivel que existen en el mercado informático, el liderazgo corresponde a un grupo reducido de ellos. En el gráfico se relacionan las características «facilidad de aprendizaje» y «número de usuarios» de los lenguajes más comunes.

dialectos más o menos especializados para una máquina particular.

h) Lenguajes especiales.

Esta categoría integra a los lenguajes que se utilizan para funciones especializadas, por lo que no son demasiado conocidos fuera de su área de trabajo. Según su aplicación pueden agruparse en distintos conjuntos como son el control de máquinas herramientas (APT, AUTOSPOT, PRONTO, ...), la ingeniería civil (COGO, STRESS, ICETRAN), diseño lógico (LOTIS, LDP), simulación (DYANA, DYNAMO, SIMULA, GPSS, SIMSCRIPT), diseño de compiladores (CLIP, TMG, META/5), análisis de microfotografías (BUGSYS), proceso y edición de textos (ES-1, SAFARI, IBMDATATEXT), salidas gráficas (DIALOG, PENCIL, GRAF), desarrollos y estudios informáticos (C), redes de datos y telecomunicación y otras muchas diferentes aplicaciones.



Los principales lenguajes de alto nivel

ADA

(En honor de Lady Augusta ADA Byron)

La publicación de unas notas sobre la máquina analítica de Charles Babbage, le sirvió a la condesa de Lovelace para pasar a la posteridad, cediendo su nombre al lenguaje que debía nacer del proyecto GREEN. Fue en 1975 cuando se consumaron los trabajos del equipo dirigido por J.M. Ischbia de la firma CII-Honeywell Bull, con el patrocinio del Departamento de Defensa de los Estados Unidos.

El ADA es un lenguaje inspirado en el PASCAL que se desarrolló con el objetivo de conseguir un lenguaje con posibilidades de convertirse en estándar universal y que facilitara el mantenimiento de los programas. Actualmente, aún no está muy difundido, aunque muchos expertos lo consideran uno de los lenguajes con mayor futuro.

ALGOL

(ALGOrithmic Languaje. *Lenguaje algorítmico)*

A raíz de un proyecto de Peter Naur en 1958, un consorcio internacional promovió el desarrollo de un lenguaje de alto nivel, inicialmente para aplicaciones científicas, que algún tiempo más tarde se plasmaría en el ALGOL. A pesar de sus cualidades para cálculos numéricos, tratamiento de entradas/salidas y procesos recursivos, el ALGOL es uno de los lenguajes que no ha viajado con la revolución microinformática.

BASIC

(Beginners All-purpose Symbolic Instruction Code. *Código de instrucciones simbólicas de uso general para principiantes)*

El más popular de los lenguajes actuales, sin lugar a dudas, y a considerable distancia de los restantes lenguajes informáticos. Nació entre 1964 y 1965 en el Dartmouth College como una herramienta para la enseñanza. Con el tiempo, han ido proliferando los dialectos y versiones, hasta el punto de que raro es el fabricante que no desarrolla un dialecto propio para sus equipos.

Es muy difícil encontrar un ordenador personal que en su versión básica no incorpore un interpréte de lenguaje BASIC. Desde hace algunos años, la firma americana Microsoft lidera el desarrollo de dialectos BASIC, disponiendo de algunas versiones (MBASIC, MSX-BASIC...) que se están convirtiendo en núcleos de estandarización.

«C»

Uno de los más recientes lenguajes de programación de alto nivel. Dentro de este marco, el «C» es uno de los lenguajes más polivalentes y próximos a la realidad de la máquina. La Bell Laboratories lo desarrolló en su origen para trabajar con el sistema operativo UNIX. La popularidad del «C» crece día a día; ello permite catalogarlo como uno de los lenguajes del futuro. Su estructura sintáctica y semántica está edificada sobre conceptos tales como estructuración, jerarquización de bloques y control del flujo de datos.

COBOL

(COmmon Business Oriented Language. *Lenguaje orientado a la gestión)*

Sin lugar a dudas, se trata del lenguaje especializado en tareas de gestión que ha alcanzado una mayor resonancia. El Departamento de Defensa de los Estados Unidos, promovió su desarrollo en 1960. A pesar de las críticas formuladas por algunos teóricos, expertos en lenguajes informáticos, su presencia es aún frecuente en miniordenadores y grandes equipos. No ocurre lo mismo en el terreno de los microordenadores; su representación en este campo se reduce a algunas versiones compiladas compatibles con el sistema operativo CP/M.

FORTH

Un lenguaje difícil de catalogar, dada su distancia respecto a los restantes lenguajes de alto nivel. A pesar de acogerse al concepto de estructuración, el FORTH mantiene una acusada proximidad respecto al lenguaje máquina; con las contrapartidas que ello supone, en cuanto a velocidad de ejecución y reducida ocupación de memoria. Otra característica reseñable es su evolutividad; el FORTH permite al usuario crear sus propios comandos. Día a día crece su proyección en el ámbito de los microordenadores. En la actualidad existen compiladores o semi-compiladores FORTH para un gran número de ordenadores personales.

FORTRAN

(FORmula TRANslation. *Conversión de fórmulas)*

Su nombre evidencia la orientación matemática de uno de los más antiguos de los lenguajes que aún predominan en nuestros días. J. Backus lo desarrolló en 1956 sobre un ordenador IBM 704. A pesar de su orientación primaria, el FORTRAN se ha revelado como un lenguaje adecuado para aplicaciones de gestión. Aunque ha perdido terreno frente a lenguajes más modernos, persiste su empleo de la mano de compiladores con versiones compatibles con sistemas operativos tan populares como el CP/M.

LISP

(LISt Processing. *Procesado de listas)*

El Massachusetts Institute of Technology creó en 1959 este lenguaje de alto nivel orientado a aplicaciones de inteligencia artificial. La programación de procesos recurrentes (edificados sobre datos sintetizados en los pasos anteriores) es uno de los puntos fuertes del LISP. Dentro de su especialidad, es un lenguaje que sigue en plena vigencia, y del que existen compiladores para microordenadores y ordenadores personales.

LOGO

Seymour Papert, del Massachusetts Institute of Technology, creó en 1976 la primera versión del popular LOGO, inspirada en su anterior desarrollo: el lenguaje LISP.

El LOGO es un lenguaje especialmente adecuado para la enseñanza asistida por ordenador. Su celebridad se debe en gran parte a la simpática «tortuga»: el símbolo con cuyo desplazamiento se generan los dibujos y presentaciones gráficas. A pesar de su popularidad, es un lenguaje encasillado en el campo educativo. Permanece ignorado por los profesionales de la programación, aunque no es desdeñable su utilidad como herramienta para la simulación de fenómenos de inteligencia artificial.

PASCAL

(En honor del célebre matemático francés Blaise PASCAL)

Este es el lenguaje estructurado por excelencia, con una presencia más que importante en el mundo de los microordenadores. N. Wirth lo desarrolló en 1969, en la escuela politécnica de Zurich, partiendo de los fundamentos del ALGOL. El PASCAL es un lenguaje muy adecuado para generar programas comprensibles y claros; ello se debe a su característica de lenguaje estructurado que obliga a la definición previa de todos los parámetros en juego.

La Universidad Californiana de San Diego, desarrolló la versión de PASCAL más popular en el campo de los microordenadores y ordenadores personales: el PASCAL UCSD.

ADA

Un paso importante hacia la unificación



os próximos párrafos van a estar dedicados a un lenguaje que, por su historia y características, se sale de la tónica que venimos observando en los lenguajes imperativos. De igual forma que una buena comida se hace partiendo de unos buenos ingredientes, en el desarrollo de ADA se han invertido los mejores recursos: decenas de cerebros y una buena cantidad de dólares.

Una historia centenaria

Comentar la historia del ADA supone remontarse al siglo XIX. Esta fue la época en la que vivió Charles Babbage, un matemático e inventor entre otras cosas del rastrillo delantero de los trenes de las películas del Oeste. Pero la gran obsesión de Babbage era la precisión matemática, lo que le llevó a idear un artilugio mecánico al que llamó «máquina diferencial» para realizar cálculos polinómicos con seis decimales de precisión, lo cual constituyó un rotundo éxito en aquel tiempo.

Posteriormente se propuso la construcción de un nuevo ingenio mecánico al que llamó «máquina analítica», el cual sería capaz de resolver cualquier operación aritmética y lógica que se le indicara a partir de unos datos en forma de tarjetas perforadas (¡un auténtico ordenador en pleno siglo XIX!). La máquina analítica nunca llegó a funcionar debido a dos razones: primero, los artesanos que trabajaban para él no eran capaces de fabricar las piezas con la precisión necesaria y, segundo, Babbage murió en el año 1871.

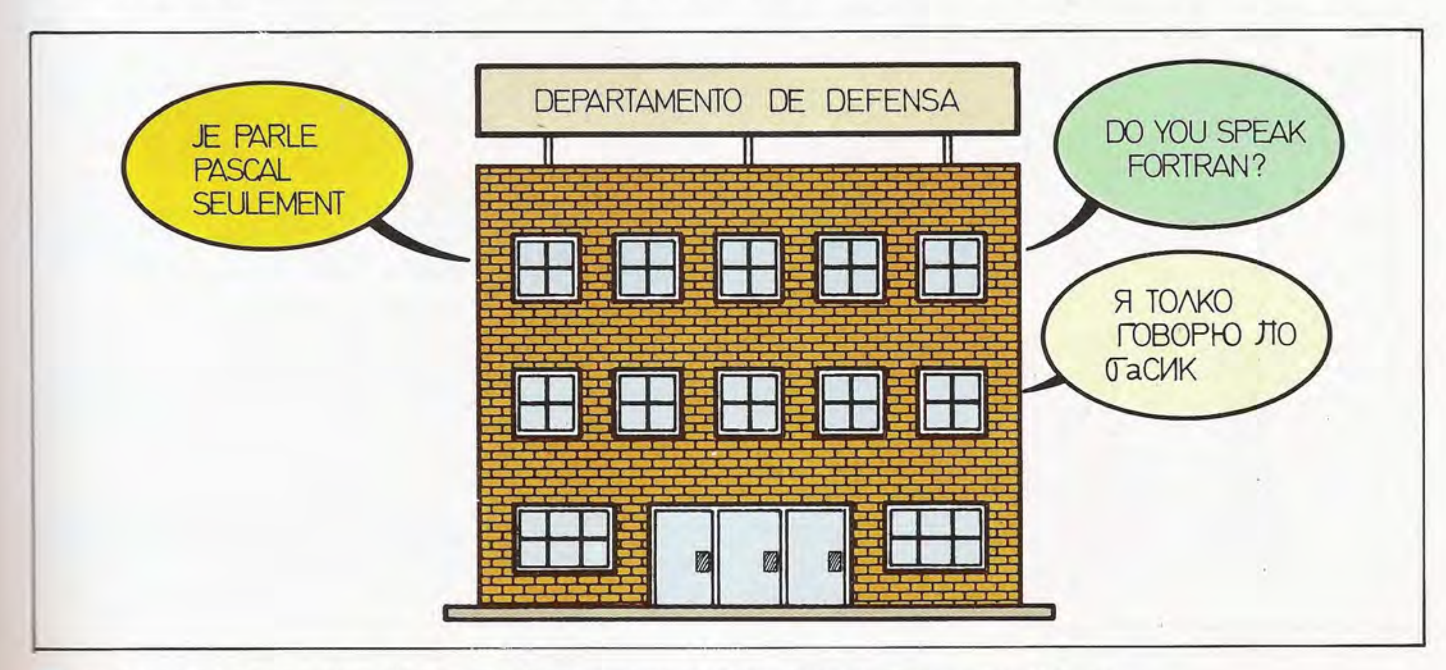
A lo largo de esta historia, Babbage recibió gran ayuda y apoyo por parte de la condesa Augusta Ada Lovelace, hija del poeta inglés Lord Byron, y cuyo promer apellido da nombre a nuestro protagonista. Ella fue la que recopiló el trabajo de Babbage a su muerte, introduciendo sus propios comentarios y notas sobre los estudios de su amigo. Gracias a su trabajo, los primeros diseñadores de ordenadores de los años cuarenta se percataron de que eran, en realidad, los segundos en llegar a la meta.

La necesidad crea el lenguaje

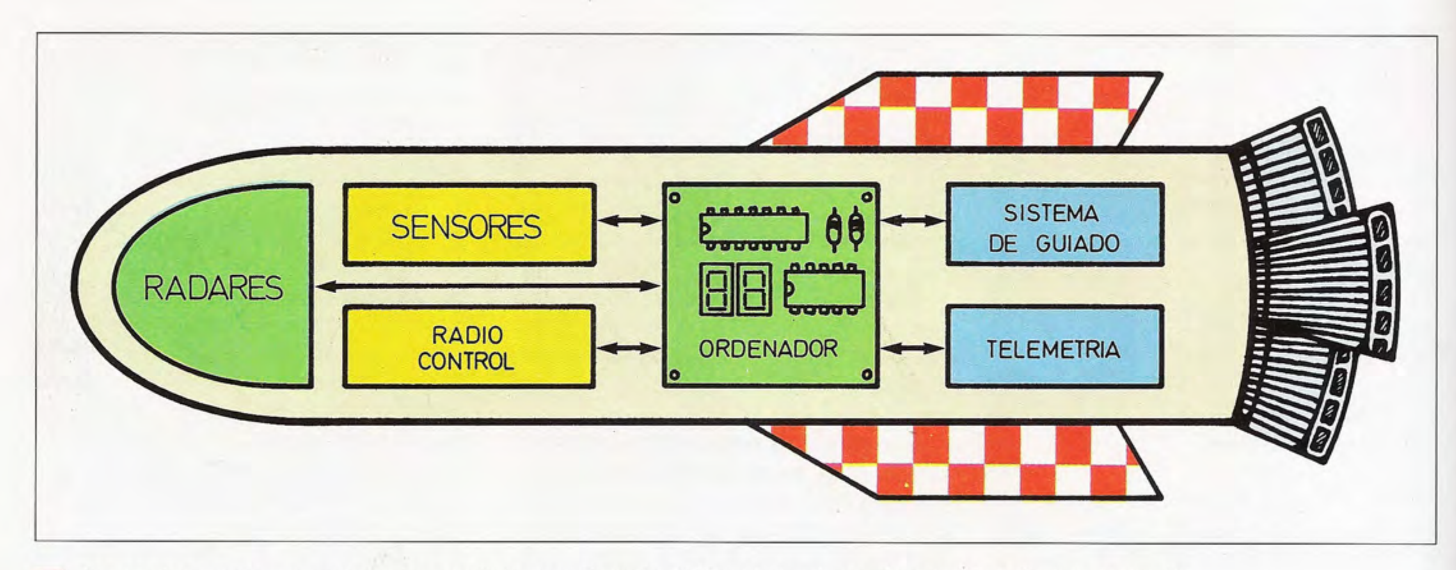
En la actualidad, el mayor consumidor de software del mundo es el Departamento de Defensa Norteamericano. La gran mayoría de sus proyectos caen dentro del grupo de los «sistemas de computadores integrados» (ver figura). Estos son sistemas mecánicos o electromecánicos en los que hay uno o más ordenadores que tienen el control sobre la mayor parte de los elementos. El éxito de las misiones del Discovery, de las sondas Pioneer o de un sofisticado avión de caza no se comprende sin tener en cuenta que detrás de ellos hay uno o más ordenadores vigilando el correcto funcionamiento de los diversos elementos. En estos ejemplos, el ordenador no es un fin, sino que está integrado en el conjunto como medio para conseguir los objetivos planteados.

ADA surgió por la necesidad de unificar los más de 400 lenguajes y dialectos que dicho Departamento utilizaba en sus proyectos, de forma que el tiempo y dinero invertidos en el desarrollo de software para uno de ellos fuera utilizable en otro de similares características.

Si analizamos las necesidades que el Departamento de Defensa puede tener, obtendremos una imagen aproximada de lo que ADA puede ofrecer. Un programa que dirija a un misil en su trayectoria puede tener más de cien mil líneas de código fuente, por lo que su desarro-



La disparidad de lenguajes utilizados en el Departamento de Defensa (DoD) norteamericano planteó la imperiosa necesidad de revisar y corregir la situación. Y de ahí nació el ADA.



En un sistema de computador integrado la coordinación del funcionamiento de todos los elementos queda confiada al ordenador u ordenadores. Este tipo de sistemas se aplica en campos tan variados como el control de misiles, la gestión de centrales telefónicas, o la coordinación y manejo de sondas interplanetarias.

llo ha de ser necesariamente una labor de equipo. El ADA proporciona las herramientas necesarias para que cada programador pueda realizar su trabajo independientemente del resto, a la vez que pueda manejar subrutinas creadas por otros sin interferir con ellas. Además, si en los cálculos que se realizan para guiar a este misil aparece una división por cero, lo más catastrófico sería que el ordenador se detuviera sin más; pensando en situaciones de este tipo, ADA también proporciona las herramientas necesarias para una gestión cómoda —y sobre todo segura— de las situaciones excepcionales que pudieran surgir en el transcurso de la ejecución de un programa.

102° 40° °F °C

```
with I_O_PACKAGE;
procedure CONVERSION_DE_TEMPERATURA is
     use I_O_PACKAGE;
     -- Este programa lee una temperatura
     -- en grados Fahrenheit y escribe su
     -- equivalente en Celsius
     GR_FAHR,GR_CEL: FLOAT;
begin
     GET(GR_FAHR);
     GR_CEL:=(5.0/9.0)*(GR_FAHR - 32.0);
     PUT(GR_CEL);
end;
```

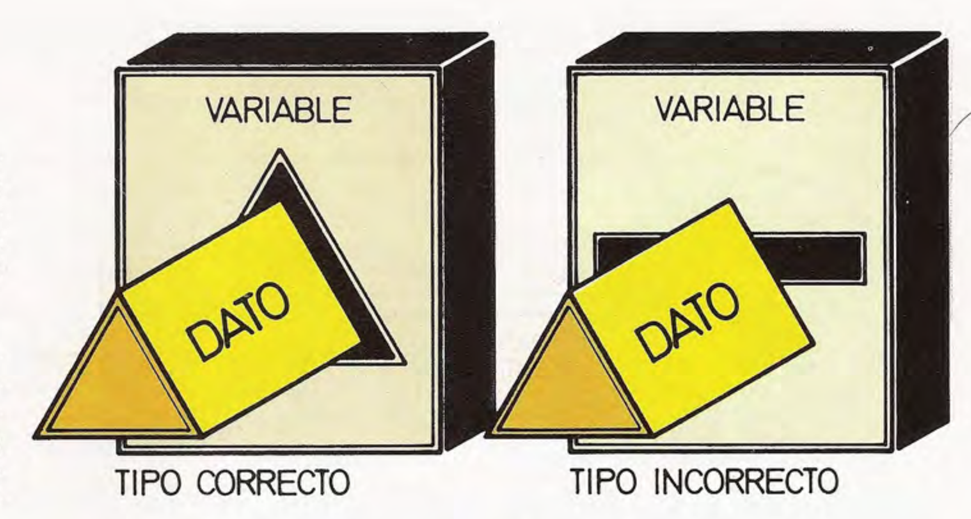
Aspecto de un programa confeccionado en ADA.

Estructura de un programa en ADA

En la figura adjunta se observa la estructura de un programa en ADA. Para los conocedores del Pascal, esta estructura resultará más que conocida. Las palabras clave están en minúsculas. El prefijo:
with I_O_PACKAGE;
es necesario ya que el programa (dedicado a la conversión de grados Fahrenheit a Celsius) utilizará los procedimientos de entrada/salida (Input/Output) PUT y GET, los cuales se encuentran en dicho paquete. Hablaremos más despacio de lo que significa un paquete.

En la figura que muestra el programa completo se observa que los comentarios van precedidos por dos guiones ("——"). Como ya hemos dicho, los procedimientos PUT y GET pertenecen al paquete de entrada/salida, y podríamos haberlos referenciado así:
I_O_PACKAGE.GET (GR_FAHR); y
I_O_PACKAGE.PUT (GR_CEL);
pero no ha sido necesario por haber incluido la cláusula:
use I_O_PACKAGE;
al principio de la parte de declaraciones.

Los ";" tienen el mismo significado que en C; esto es, como terminadores de sentencias en vez de separadores de las mismas como en Pascal. La declaración:
GR_FAHR,GR_CEL: FLOAT;
declara a estas variables como de tipo real (valores en punto flotante).



Las variables declaradas con un tipo específico sólo pueden almacenar valores de ese tipo. Cualesquiera otros valores serán rechazados mediante el oportuno mensaje de error.

```
with I_O_PACKAGE;
procedure CONVERSION_DE_TEMPERATURA is
    --parte de declaraciones
begin
    --sentencias
end;
```

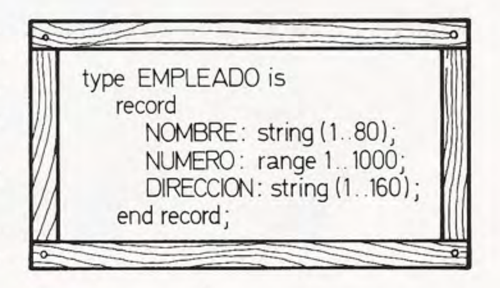
La estructura de un programa escrito en ADA es muy similar a la de otro codificado en Pascal. No en vano este último fue tomado como uno de los modelos a partir de los cuales se generó ADA.

Los tipos en ADA

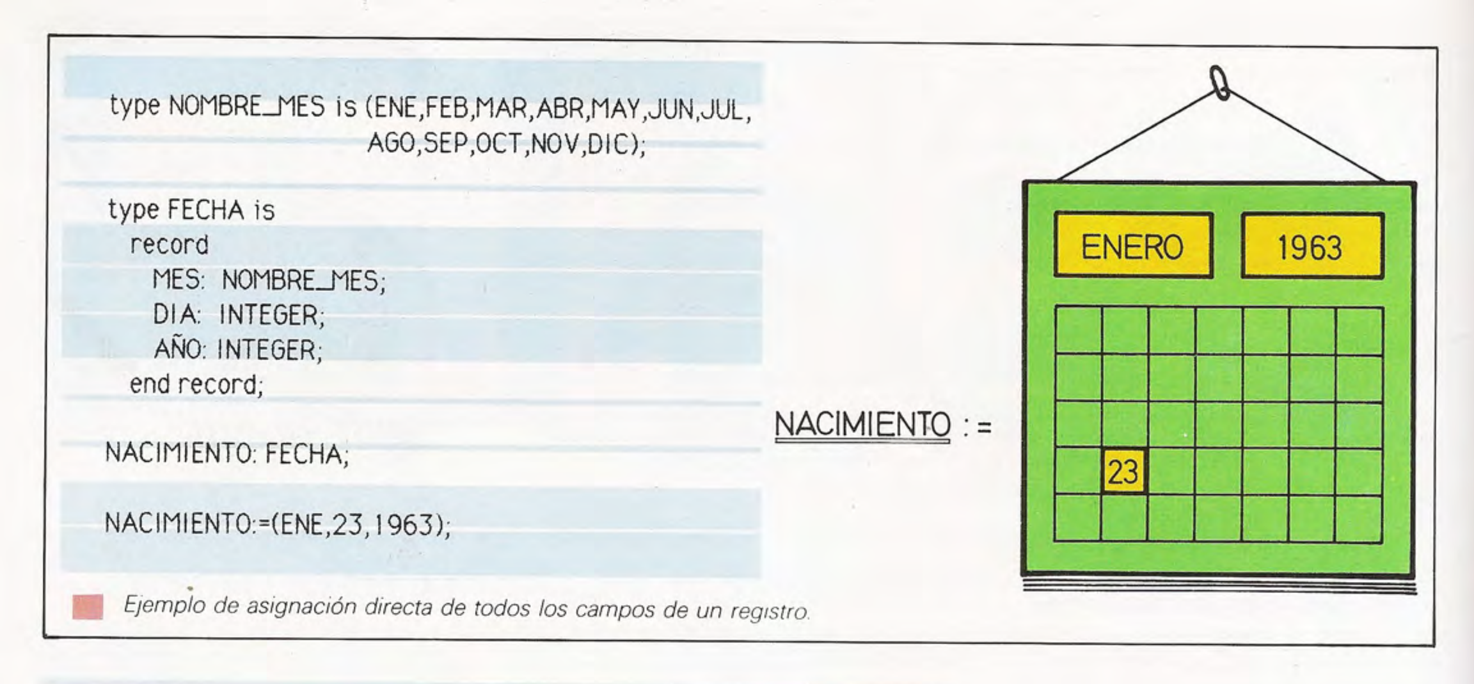
El ADA comparte con el Pascal la característica de ser un lenguaje «tipado»; esto es, al igual que existen unos tipos de datos básicos, como enteros (INTEGER), reales (FLOAT), etc., el usuario puede definir sus propios tipos.

Uno de los objetivos prioritarios del ADA es ser un lenguaje seguro, en el que nada se escape al programador en el momento de la ejecución. Así, si tenemos una variable "DIA" que representa el día de la semana en que estamos, sería estúpido que tomara el valor —5.3E01. Es deseable que el lenguaje rechace la asignación a variables de valores que no tienen sentido, como el anterior. Todo esto redunda en la confección de programas más seguros. En nuestro caso, definiríamos el tipo "DIAS_DE_LA_SEMANA" así:
type DIAS_DE_LA_SEMANA is (LUN,MAR,MIE,JUE,VIE,SAB,DOM);
e indicaríamos que "DIA" sólo puede tomar estos valores de la siguiente forma:
DIA: DIAS_DE_LA_SEMANA;

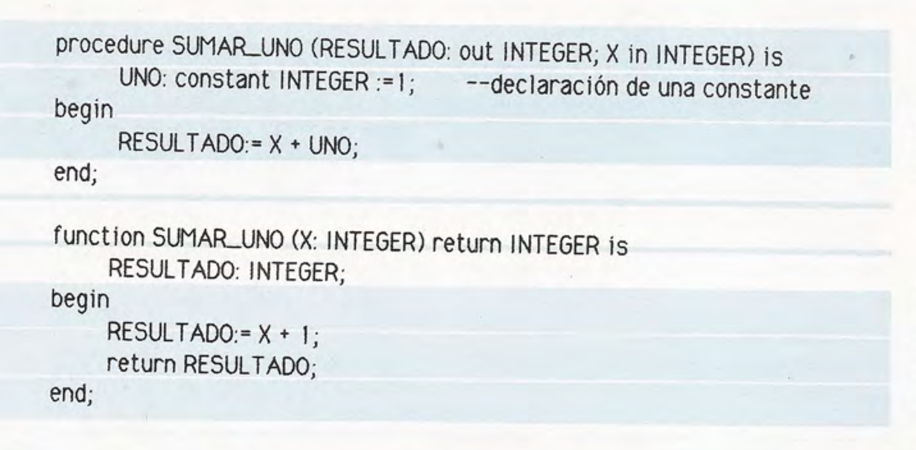
Tal definición tendrá lugar en la parte de declaraciones, de igual forma que se hace en Pascal.



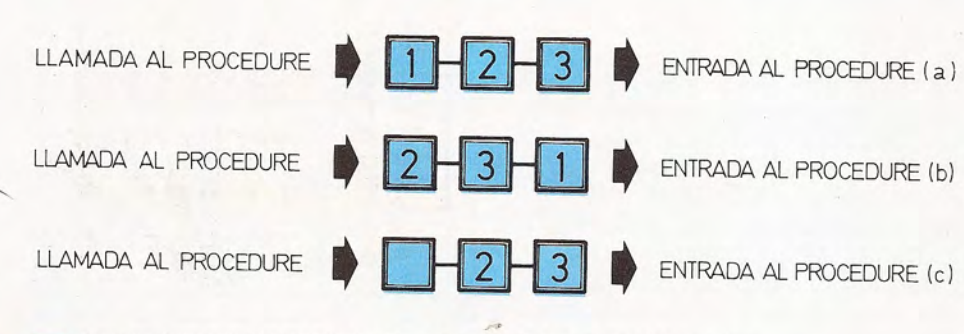
Los registros permiten agrupar una serie de variables de tipos distintos dentro de un marco común.



Ejemplo de asignación directa de todos los campos de un registro.



Los procedimientos y funciones son las herramientas básicas de que dispone el usuario de ADA para estructurar sus programas.



Los parámetros destinados a los procedimientos del ADA pueden ser pasados tal y como es costumbre hacerlo en PASCAL (a), cambiando su orden (b), o incluso omitiendo alguno de ellos, si en la declaración se han consignado valores por defecto (c).

Estructuras de control

Las estructuras de control del ADA son las habituales en otros lenguajes estructurados; están resumidas en el cuadro adjunto.

- for-loop

Equivalente al FOR-DO del Pascal. El bucle que aparece en el ejemplo se ejecutará para valores de "CONTADOR" desde 1 hasta 10. Presenta la particularidad en ADA de que no es necesario declarar la variable del bucle en la parte de declaraciones. Consecuentemente, el ejemplo no declara a "CONTADOR" como variable entera.

- while-loop

Plena similitud con su equivalente en Pascal.

- loop-end loop

Esta estructura no se corresponde con ninguna del Pascal. Simplemente permite ejecutar reiteradas veces una serie de sentencias. Si no se desea permanecer dentro del bucle indefinidamente, será preciso introducir un "exit when", el cual hará que el control pase a la primera sentencia que sigue a "end loop" cuando la condición entre paréntesis sea cierta. En nuestro caso (ver cuadro)

habríamos declarado a "FINAL" como de tipo BOOLEAN y su valor sería actualizado en alguna sentencia del bucle.

- if-then

Prácticamente como en Pascal, a excepción del ";" que precede a "else". Hay que recordar el distinto significado de este símbolo comentado anteriormente.

- case-is

En el ejemplo se hace referencia a la variable "DIA" cuyo tipo está definido en el párrafo anterior. Su efecto será escribir "Hoy es fiesta" si "DIA" es DOM ó SAB, y "Hoy no es fiesta" en los restantes casos.

Si hay que sacar alguna conclusión de todo esto, no puede ser otra que el triunfo de las tesis sobre programación estructurada que sentó el lenguaje Pascal diez años antes que el ADA.

En el caso del ADA, hay que seguir a la palabra clave "end" con otra que refleje a qué final nos estamos refiriendo. Los que hayan programado en Pascal pueden haberse encontrado en ocasiones con una ristra de ENDs de los que, en el caso de no haber seguido una norma de indentación, no se sabe a ciencia cierta a qué estructura pertenecen.

Seguiremos viendo más coincidencias con el Pascal, redundando en la anterior afirmación. Se comprobará que las diferencias sintácticas que aparecen en el ADA están orientadas a facilitar la comprensión de los listados.

Estructuras de datos

Todo lenguaje estructurado que se precie —y el ADA lo es— debe permitir al programador representar en su programa los datos de la realidad de una forma fiel y cómoda de manejar. Esto irá en beneficio de una mayor corrección y facilidad a la hora de realizar posibles cambios en los programas.

Al igual que en el Pascal, en ADA es posible manejar ARRAYs y RECORDs, y con estos últimos construir listas encadenadas y árboles binarios. Así, por ejemplo, la siguiente línea:

```
B: array (1..100) of INTEGER;
```

declara un vector (B) de cien elementos enteros.

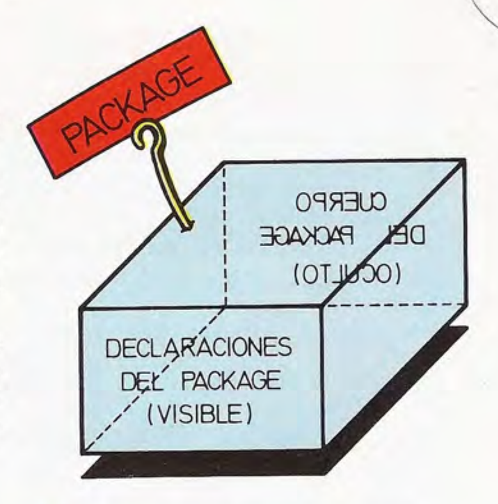
ADA proporciona potentes herramientas para la inicialización de arrays sin tener que realizar bucles, manejar secciones de los mismos, e incluso hacer declaraciones de arrays cuya dimensión máxima esté sin especificar. Los arrays de caracteres se llaman «string» y se declaran como sigue:

```
PALABRA: STRING (1..10);
```

y ahora podríamos hacer cosas como:

```
PALABRA:="HOLA"
```

El ADA también incluye operadores especiales para tratar los arrays de caracteres.



El usuario de un «package» sólo puede acceder a la parte visible del mismo. Su cuerpo (body) queda oculto por las declaraciones.

Los RECORDs son en ADA muy similares a como lo son en Pascal. En la figura se observa un ejemplo de declaración de un registro que se podría utilizar en la gestión de una empresa de no más de 1000 empleados; el equivalente en Pascal de la declaración de la variable "NUMERO" sería:

```
NUMERO: 1..1000;
```

Los RECORDs pueden tener estructuras alternativas. (Los parientes pascalianos de este tipo de registros son los RECORDs variantes). El acceso a los cam-

```
package ESTADISTICA is

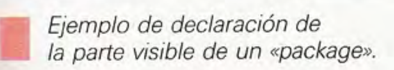
    PI: constant FLOAT :=3.141592654;
     E: constant FLOAT :=2.718281828;

    procedure FACTORIAL (N: in INTEGER; FACT: out INTEGER);

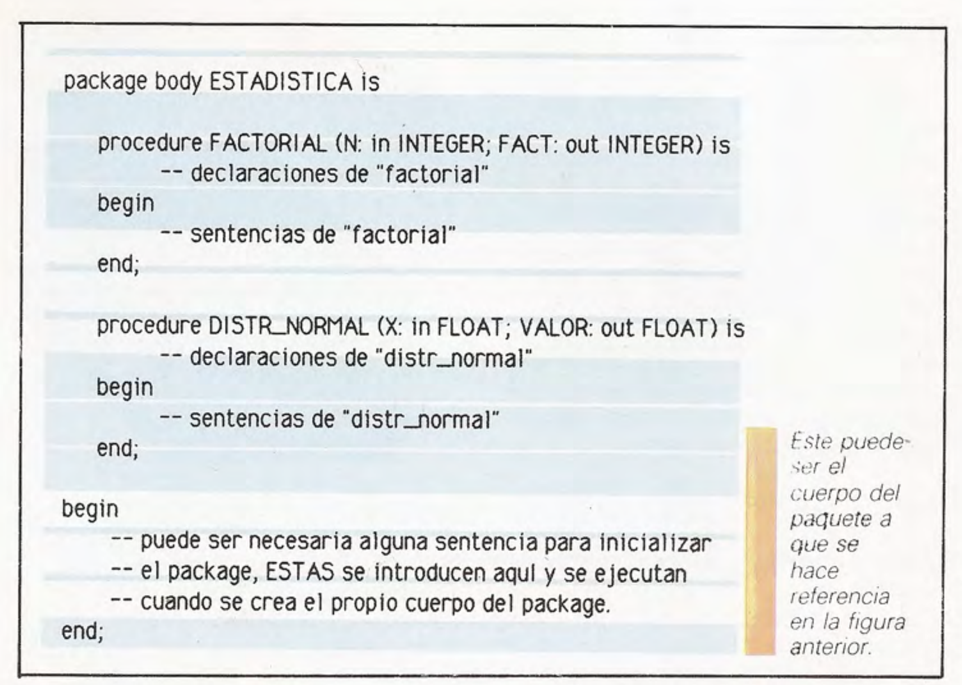
    procedure DISTR_NORMAL (X: in FLOAT; VALOR: out FLOAT);

end;
```





Ejemplo de declaración de la parte visible de un «package».



```
package body ESTADISTICA is

    procedure FACTORIAL (N: in INTEGER; FACT: out INTEGER) is
            -- declaraciones de "factorial"
    begin
            -- sentencias de "factorial"
    end;

    procedure DISTR_NORMAL (X: in FLOAT; VALOR: out FLOAT) is
            -- declaraciones de "distr_normal"
    begin
            -- sentencias de "distr_normal"
    end;

begin
      -- puede ser necesaria alguna sentencia para inicializar
      -- el package, ESTAS se introducen aqui y se ejecutan
      -- cuando se crea el propio cuerpo del package.
end;
```

Este puede ser el cuerpo del paquete a que se hace referencia en la figura anterior.

pos del registro se realiza también a través del operador ".". Como muestra la correspondiente figura, podemos inicializar un registro completo en una sola sentencia, sin necesidad de tener que hacerlo campo a campo.

Procedimientos y funciones en ADA

Son equivalentes a sus homónimos del Pascal, aunque en el ADA tienen nuevas características que los/las hacen más potentes. En la figura adjunta se tiene una declaración de cada tipo.

Los parámetros que se pasan a un procedimiento pueden ser de tres tipos:

- tipo "in"

Constituyen la entrada al procedimiento. Dentro de él son como constantes y sus valores no pueden ser modificados.

- tipo "out"

Son la salida del procedimiento

- tipo "in out"

Son considerados como variables cuyos valores se pueden modificar durante la ejecución del procedimiento, y estas modificaciones quedan reflejadas a la salida del mismo.

La declaración de "UNO" que tiene lugar en el procedimiento ilustrado sería equivalente en PASCAL a:

```
CONST UNO=1;
```

En cuanto a las funciones, éstas sólo admiten parámetros del tipo "in" (aun-

$\vec{X} = (X_1, X_2, X_3)$ $\vec{Y} = (Y_1, Y_2, Y_3)$ $\vec{X} \cdot \vec{Y} = X_1 \cdot Y_1 + X_2 \cdot Y_2 + X_3 \cdot Y_3$

S:= 0
$S := S + X_1 \bullet Y_1$
$S := S + X_2 \bullet Y_2$
$S := S + X_3 \bullet Y_3$
CALCULO SECUENCIAL

$S_1 := X_1 \bullet Y_1$ $S_2 := X_2 \bullet Y_2$ $S_3 := X_3 \bullet Y_3$
$S := S_1 + S_2 + S_3$
CALCULO SIMULTANEO

Dos formas de realizar en ADA el producto escalar entre vectores.

que no es necesario decirlo explícitamente como ocurre en el ejemplo). El uso de las funciones es análogo al que se hace en Pascal; apareciendo sus nombres en expresiones sintácticamente correctas y cuidando la compatibilidad del tipo devuelto.

Pero en ADA los procedimientos y funciones dan todavía más que hablar. Una llamada a "SUMAR_UNO" podría consistir en:

```
SUMAR_UNO(INCREMENTADO,
NUMERO);
```

la cual se denomina «llamada posicional»: los parámetros de la llamada se sustituyen en la declaración en el mismo orden. La alternativa que ofrece el ADA consiste en hacer:

```
SUMAR_UNO(X=>NUMERO,
RESULTADO=>INCREMENTADO);
```

Con ello especificamos el paso de los parámetros explícitamente, rompiendo la norma posicional de la llamada anterior.

Además, los parámetros del tipo "in" pueden tomar valores por defecto. Si cambiáramos la cabecera de "SUMAR_UNO" por:

```
procedure SUMAR_UNO(X: in INTEGER:=5; RESULTADO:
out INTEGER)
```

una llamada podría consistir ahora en:

```
SUMAR_UNO(INCREMENTADO);
```

y al acabar la ejecución del procedimiento la variable "INCREMENTADO" tendría el valor 6 (ver figura).

Los paquetes (packages)

En un apartado precedente se mencionó que las aplicaciones para las que está concebido el lenguaje ADA pueden ser programas de enorme longitud.

En esta categoría de programas, la posibilidad de estructurarlos a base de procedimientos y funciones puede resultar insuficiente, ya que su complejidad puede desbordar la capacidad de «modularización» que proporcionan las citadas estructuras.

En ADA existe otra forma —no excluyente con la conocida— de estructurar los programas: el «package». En uno de ellos se pueden incluir una serie de tipos, datos, procedimientos y funciones que lógicamente estén relacionados. En términos generales, un package ofrece un servicio; y como cualquier servicio, puede precisar de otros datos y servicios internos que no son visibles al usuario del servicio original. Un package se compone de dos partes: la parte visible y el propio cuerpo del package. Sólo la primera será accesible a la persona que quiera utilizarlo.

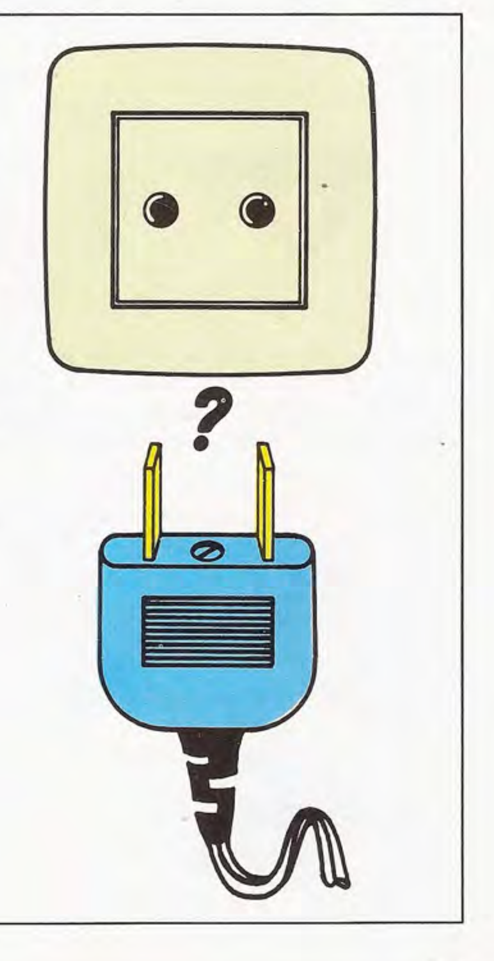
Suponga que se trata de redactar un programa que necesita algunas funciones matemáticas que ADA no proporciona directamente. Para empezar, estructuraremos la parte principal en forma de package; de esta forma nuestro trabajo será útil a otra persona que pueda necesitarlo en el futuro. En la figura se observa el posible aspecto de la parte visible de nuestro package; es decir, aquella que podremos utilizar en el programa. En ella realizamos la declaración de

Lenguajes imperativos y declarativos

Como ya se ha comentado al principio, el ADA puede considerarse el exponente más avanzado de los lenguajes imperativos. Este tipo de lenguajes se caracteriza porque, con ellos, se le indican al ordenador de forma inequívoca los pasos a seguir para la resolución de un problema: se trata de expresar un algoritmo en términos comprensibles para el ordenador. Con los avances en los estudios sobre inteligencia artificial y los nuevos ordenadores de la quinta generación, han hecho aparición otros lenguajes cuya filosofía es distinta: se trata de los lenguajes declarativos. Con ellos se introducen al ordenador unas reglas generales sobre la forma en la que debe resolver el problema y manejar los datos. Para llevar a cabo su tarea, el ordenador seleccionará aquellas reglas que se le «antojen» adecuadas en cada momento.
Cabe resumir diciendo que con un lenguaje declarativo se le dice al ordenador *qúe* debe hacer, mientras que con un lenguaje imperativo se le dice *cómo* debe hacerlo.

La portabilidad del ADA

Los lenguajes a los que ADA va a sustituir son utilizados sobre ordenadores muy distintos, con diversos tipos de procesadores y mecanismos de entrada/salida también diferentes. Invertir la cantidad de recursos que se han invertido sin tener en cuenta este hecho sería una locura. Por ello, hay que añadir a las características ya comentadas del ADA su portabilidad; esto es, la posibilidad de que se puedan ejecutar en distintos ordenadores programas escritos en este lenguaje. Normalmente, los mayores problemas de portabilidad se centran en la distinta configuración de los mecanismos de entrada/salida de los ordenadores. La posibilidad de estructurar los programas en paquetes hace que, en la mayoría de las ocasiones, baste con modificar el I_O_PACKAGE para adaptar el programa al ordenador en cuestión.



dos constantes y dos procedimientos: uno para el cálculo del factorial de un número y otro para calcular la función de distribución normal estándar. Estos elementos será posible emplearlos en aquellos programas a los que se indique que utilicen el package "ESTADISTICA", como se verá más adelante.

En una de las figuras aparece tan sólo la parte visible. En el cuerpo (body) del package será preciso realizar la declaración completa de los procedimientos parcialmente declarados en la parte visible. Ello se hace tal como ilustra la figura siguiente. Lo comentado al principio acerca del I_O_PACKAGE es aplicable a cualquier otro paquete. Si en nuestro programa quisiéramos utilizar los datos o procedimientos definidos en el paquete "ESTADISTICA" añadiríamos al principio del mismo:

```
use ESTADISTICA;
```

y de esta forma eliminaríamos la necesidad de tener que referenciar a "FACTORIAL" así:

```
ESTADISTICA.FACTORIAL(X,Y);
```

Ahora, bastaría una llamada tipo:

```
FACTORIAL(X,Y);
```

Procesamiento en paralelo

La posibilidad de controlar dos o más eventos cuyas acciones ocurren simultáneamente tiene gran importancia en el tipo de trabajos para los que ADA está pensado. Suponga que se están dirigiendo dos misiles hacia un objetivo concreto. En esta situación sería muy interesante poder controlar los dos misiles simultáneamente, de forma que si uno de ellos ha conseguido su objetivo se pueda desviar al que queda hacia un nuevo objetivo.

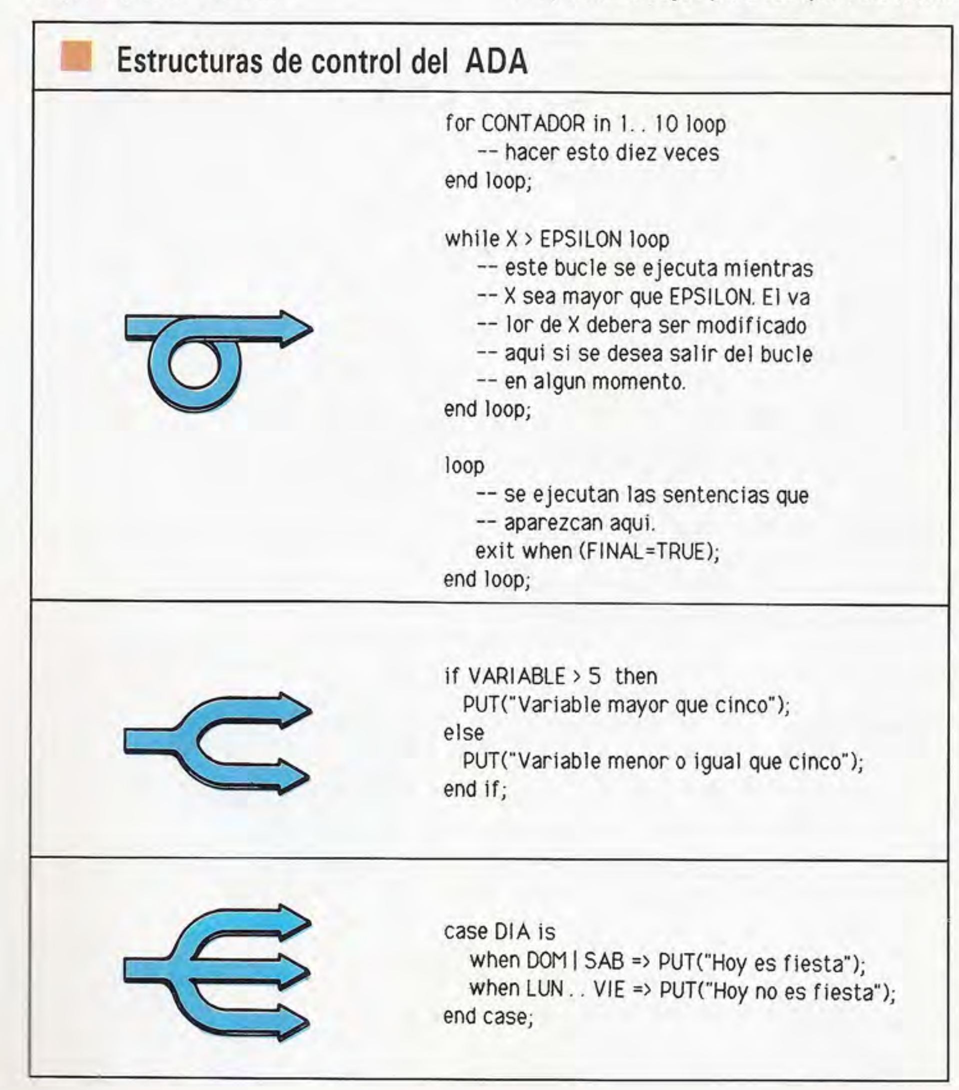
Bajando a terrenos menos conflictivos, suponga ahora que se desea calcular el producto escalar de dos vectores de tres coordenadas (ver figura). La única forma que conocemos hasta ahora para realizar este cálculo es efectuar las sumas y los productos *secuencialmente*, como refleja la primera zona de la mencionada figura. Sin embargo, en ADA puede hacerse que los productos de las coordenadas se realicen *simultáneamente* y, una vez obtenidos estos valores, sumarlos en el siguiente paso. El beneficio en tiempo de ejecución para un vector de tres coordenadas no compensa la complejidad adicional de realizar un procesado en paralelo; pero si el vector fuera de 1000 coordenadas, un enfoque como el descrito puede resultar muy ventajoso.

Estas posibilidades «software» que nos proporciona el ADA están basadas en los adelantos hardware de los últimos años. Un ordenador que pueda soportar eficientemente el tipo de cálculo expresado más arriba ha de poseer más de una CPU, de forma que cada una de ellas se ocupe, con simultaneidad, de un proceso.

El uso de procesos paralelos para la resolución de un problema trae consigo a menudo muchos quebraderos de cabeza.

Volviendo a nuestro ejemplo, una CPU no tardará lo mismo en realizar el producto 1×1 que el producto 1543.43×12E5, por lo que la necesidad de sincronizar las diversas tareas encomendadas a cada procesador es una necesidad evidente. Además, hay que contemplar la posibilidad de que se comentan errores en alguna CPU mientras se están realizando los cálculos, lo que complica aún más tanto el diseño del hardware como del software.

En ADA los procesos paralelos se expresan mediante «task» (tareas), cuya apariencia es muy similar a la de los packages. Las «tasks» están acompañadas por una serie de sentencias que facilitan su sincronización.



C (1)

Un lenguaje para la programación de sistemas



Los orígenes del C se remontan al año 1972, en los laboratorios de la compañía estadounidense Bell. Contrariamente a la tendencia actual, según la cual los grandes proyectos informáticos son encomendados a un equipo de profesionales, C surgió de la mente de un solo hombre, Dennis Ritchie, que por aquel entonces trabajaba en la citada compañía.

No es posible hablar de los orígenes de C sin hacer mención del sistema operativo UNIX. Los años 70 fueron testigos del nacimiento de UNIX, el cual, posteriormente, se convierte en un estándar «de facto» (una norma que, sin tener carácter de ley, es aceptada casi universalmente) para los sistemas operativos mutiusuario en el campo de los grandes ordenadores. UNIX proporciona a sus usuarios una serie de herramientas para ayudar a la confección de programas; C



El C es un lenguaje de programación especialmente pensado para su uso en el desarrollo de software de sistemas.

1960 — ALGOL 60 (ALGORITHMIC LANGUAGE)
DISEÑADO POR UN COMITE INTERNACIONAL

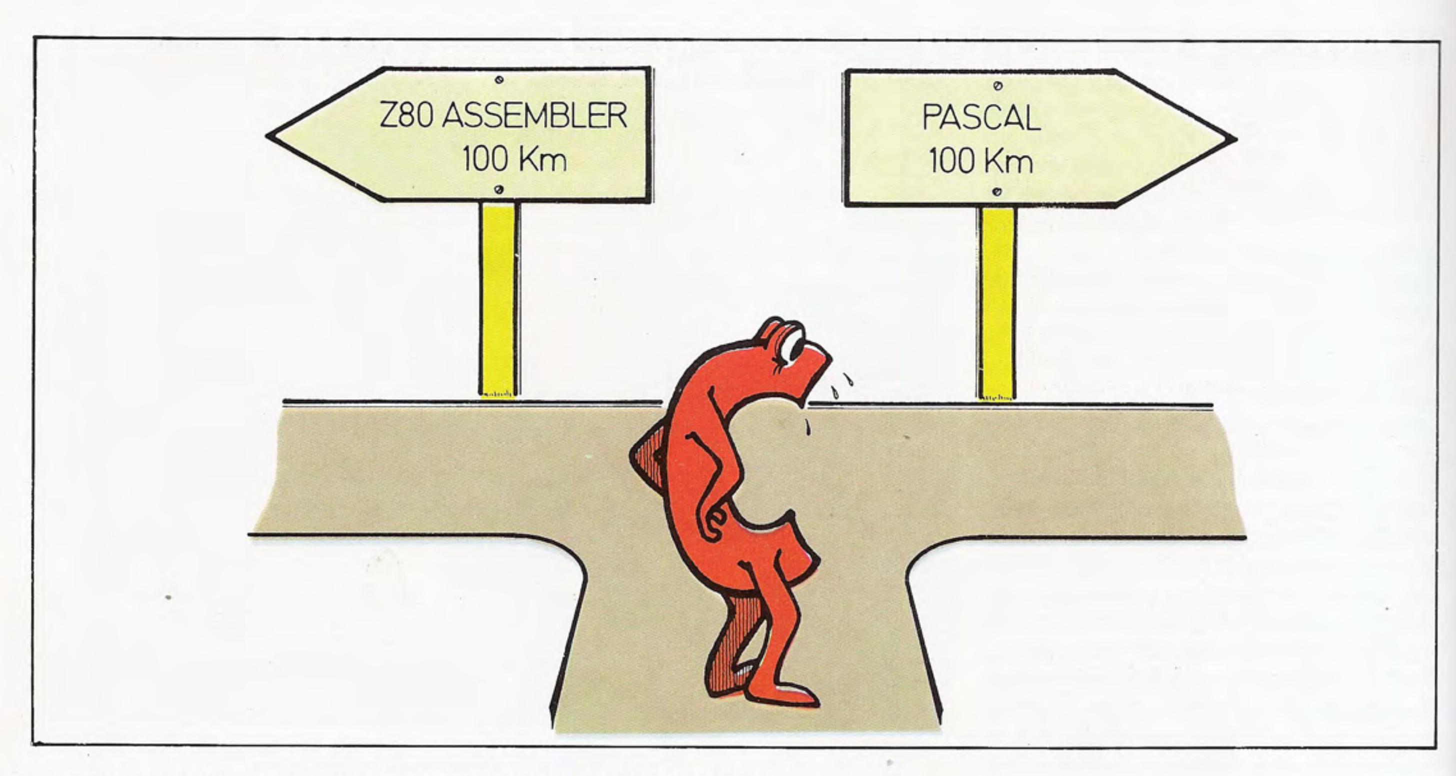
1963 — CPL (COMBINED PROGRAMMING LANGUAGE)
UNIVERSIDADES DE CAMBRIDGE Y LONDRES

1967 — BCPL (BASIC COMBINED PROGRAMMING LANGUAGE)
MARTIN RICHARDS (UNIV. DE CAMBRIDGE)

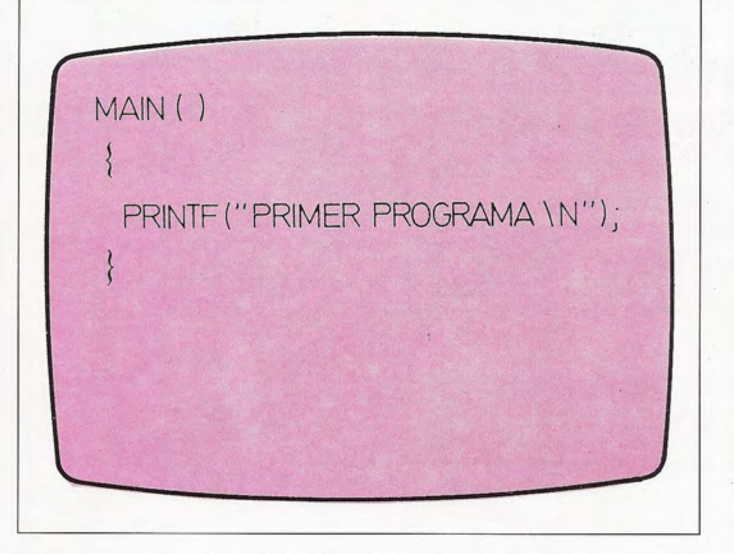
1970 — B
KEN THOMPSON (LABORATORIOS BELL)

1972 — C
DENNIS RITCHIE (LABORATORIOS BELL)

Hasta la definición del C como hoy lo conocemos fue preciso pasar por diversos lenguajes más o menos potentes.



El lenguaje C se encuentra a mitad de camino entre los lenguajes de alto nivel y los lenguajes ensambladores.



Ejemplo de programa confeccionado en C. En la pantalla se observa que el C es un lenguaje de los denominados «de formato libre».



Algunas secuencias de escape.

era una de esas herramientas, y muy pronto se convirtió en «la» herramienta, hasta el punto de que el propio compilador de C y la mayor parte de UNIX se reescribieron en dicho lenguaje. Vemos ya aquí un aspecto de C sobre el que volveremos más adelante: su capacidad para la creación de software de sistemas.

La historia del C

La historia del lenguaje C es un modelo de claridad y concisión con el que nos gustaría encontrarnos más a menudo. En la figura adjunta se refleja el proceso que llevó a la creación del C.

Algol es el segundo «gran lenguaje» de programación desarrollado en la era informática. Fortran fue el primero, y Algol intentó mejorarlo en sus muchos puntos débiles, cuidando especialmente la sintaxis y proporcionando una estructura modular. El resultado fue un lenguaje demasiado abstracto y general, por lo que nunca disfrutó de gran aceptación. CPL intentó, sobre las bases de Algol, traer las cosas más próximas a la realidad; de cualquier manera, Algol todavía estaba demasiado presente, por lo que el nuevo lenguaje seguía siendo difícil de aprender e implementar. BCPL intentó solucionar estos problemas extrayendo las características básicas de CPL.

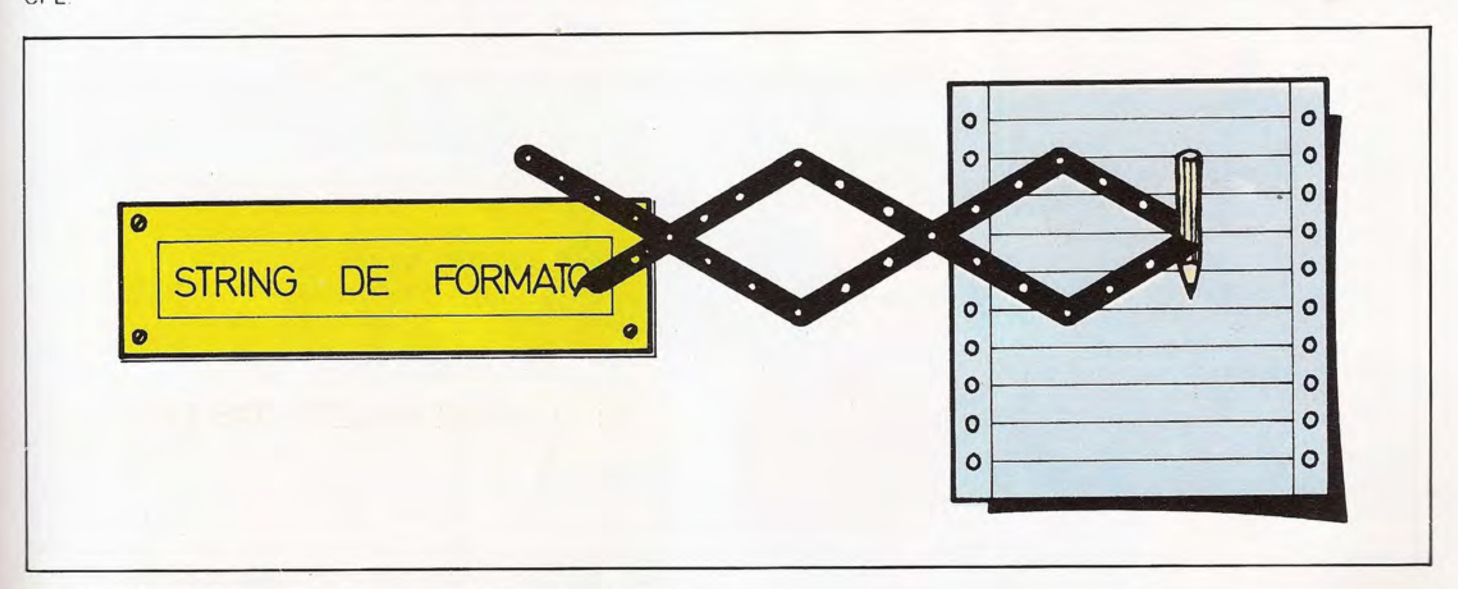
PRINTF (STRING DE FORMATO , ARGUMENTO , ARGUMENTO ,.....);

Formato general de la función «printf».

```
main ( )
{
int b;
int a=1;

b=mi__función (a);
printf ("El resultado de mi función es %d\n",b);
}
mi__función (i)
int i;
{
return (i + 1);
}
```

Programa ejemplo que ayudará a profundizar en el lenguaje C.



El «string de formato» es la pauta según la cual serán escritas las variables presentes en la instrucción «printf».

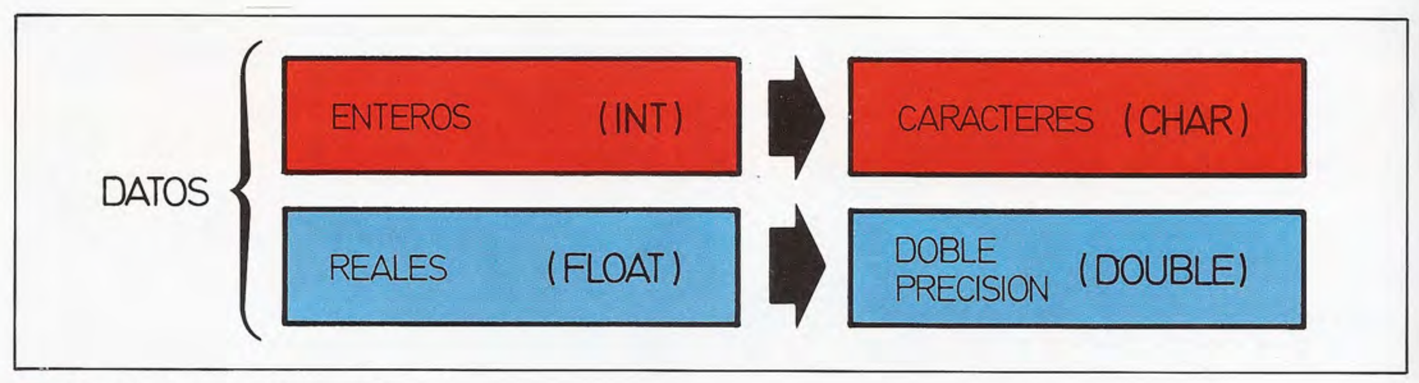
Siguiendo la misma línea, B es un nuevo refinamiento de BCPL.

El resultado de esta evolución fue que los lenguajes eran cada vez más fáciles de implementar y de aprender, a costa de haberlos restringido a dominios de aplicación cada vez más estrechos. La virtud de C está en dar un paso atrás en la trayectoria mencionada, añadiendo algo más de generalidad y manteniendo casi el mismo nivel de dificultad.

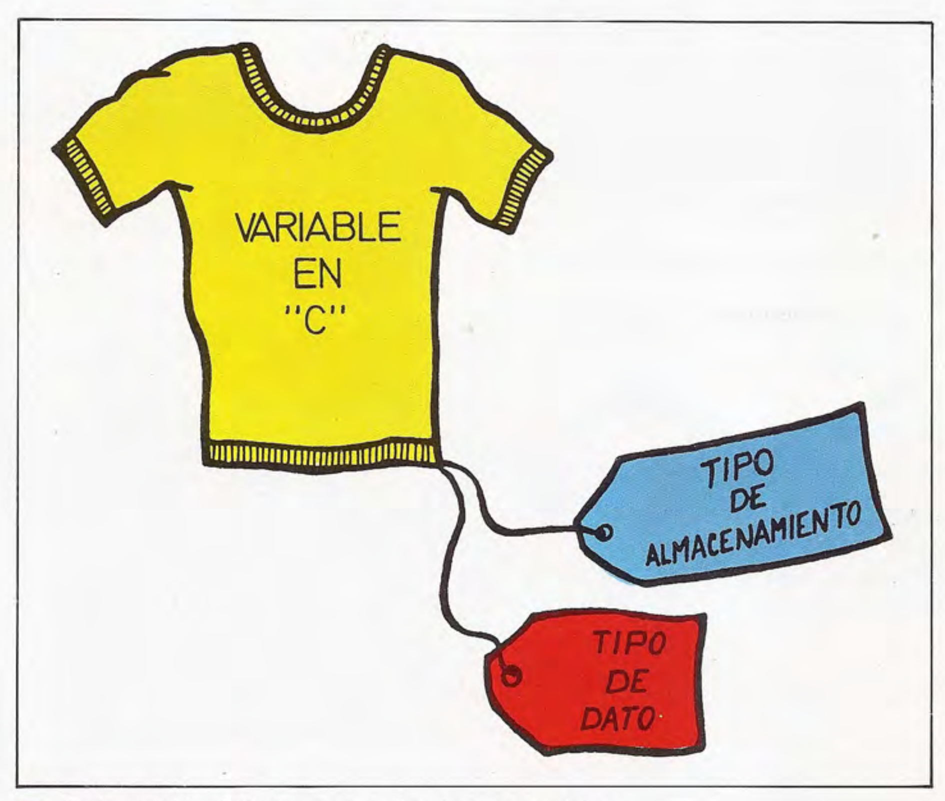
¿Qué es el C?

Como ya se ha comentado anteriormente, C está especialmente pensado para su uso en el desarrollo de software de sistemas. El software de sistemas enlaza al hardware con el usuario final del equipo. El primer ejemplo de software de sistemas es el propio sistema operativo.

Esto no quiere decir que no sea posible hacer un programa de juegos en C (prácticamente se puede hacer casi todo



Tipos de datos en el lenguaje C.



Toda variable de C tiene asignadas dos características.

con cualquier lenguaje), sino que, simplemente, sacaremos más rendimiento a nuestro tiempo aplicando C al desarrollo de software de sistemas.

C es un lenguaje que permite estar lo suficientemente próximos a la máquina como para manejar directamente posiciones de memoria, bits aislados dentro de estas posiciones, e incluso, de alguna forma, los propios registros de la CPU; a la vez que proporciona las estructuras características de los lenguajes de alto nivel, tan importantes para hacer de la programación una tarea no tan difícil. Por todo esto, cabe afirmar que el C se encuentra a mitad de camino entre un ensamblador y un lenguaje de alto nivel de tipo estructurado como el PASCAL.

El camino hasta ejecutar un programa en C

A pesar de que existen en el mercado algunos intérpretes de C, lo normal es que este lenguaje sea compilado. En la figura se observan los pasos que transcurren desde la creación del programa por medio del editor, hasta conseguir un fichero con el programa listo para ser ejecutado.

Uno de los objetivos de Dennis Ritchie al concebir el lenguaje C era la simplicidad y sobre todo la portabilidad. Por tal se entiende el que un programa en C pueda ser ejecutado en máquinas totalmente diferentes, con distinta arquitectura y configuraciones de entrada/salida. Como ya se ha comentado, el propio sistema operativo UNIX está escrito mayoritariamente en C, y en la actualidad, fabricantes tan diversos como Digital, IBM o Honeywell incorporan como sistema operativo a UNIX.

C no posee instrucciones específicas de entrada/salida. Una vez que el ensamblador ha generado el código objeto quedarán lo que técnicamente se denominan «referencias sin resolver», que son básicamente llamadas a subprogramas que realizan las operaciones de entrada/salida. El «linker» tomará dicho código objeto y accederá a los ficheros donde se encuentran las referencias, incluyendo en el nuevo código generado

los segmentos de programa necesarios para «resolver» tales referencias.

Aparencia de un programa en C

La estructura fundamental a partir de la cual se forman programas en C es la función. Todo programa debe tener al menos una de ellas y siempre habrá una que se llame «main» —principal en inglés—. Pueden estar repartidas en cualquier orden a lo largo del programa y la ejecución empezará siempre por «main»; aunque no hay ninguna regla sintáctica que nos obligue a ello, se suele colocar a «main» al principio del listado.

```
mi_función (i)
int i;
{
int cte;
cte=3;
return (i+cte);
}
```

Programa en C en el que se hace uso de una variable automática (cte), que sólo puede ser referenciada dentro del campo de visión de "mi_función".

```
1)
main ()
{
incrementar;
incrementar;
incrementar;
}
incrementar ()
{
int x=0;
x=x+1;
printf("%d\n",x);
}

SALIDA:
1
1
1
—
```

```
2)
main ()
{
incrementar;
incrementar;
incrementar;
}
incrementar ()
{
static int x=0;
x=x+1;
printf("%d\n",x);
}

SALIDA:
1
2
3
—
```

En la parte (2) de este ejemplo, la variable "x" ha sido declarada como estática, por lo que su valor sigue existiendo después de haber abandonado "incrementar". Obsérvese que la inicialización de "x" sólo ocurre en la primera pasada por la citada función.

C es un lenguaje de los llamados «de formato libre»; esto quiere decir que, al igual que en PASCAL, los espacios en blanco no tienen significado, y son utilizados para resaltar la estructura del programa. Los comentarios van entre los grupos/* y */. En la figura adjunta aparece un ejemplo de programa en C. Como vemos, está presente «main» seguido de un par de paréntesis. Toda función tendrá unos argumentos sobre los que se elaborará el resultado de la función. En este caso, la función «main» no tiene ninguno, por lo que la lista de argumentos está vacía; los paréntesis, sin embargo, siguen siendo obligatorios. Las llaves ({}) delimitan el cuerpo de la función y son equivalentes a los BEGIN-END de Pascal.

```
char x='f';
main ()
{
printf ("%c\n",x);
}
```

Programa ejemplo en el que se hace uso de una variable externa.

En el cuerpo de «main» nos encontramos una sentencia de salida, la cual escribe por pantalla el argumento que tiene entre comillas dobles. Por pertenecer esta sentencia al grupo de entrada/salida, NO es parte de C; podría haberse llamado de cualquier otra forma que le hubiera gustado al implementador del C de nuestro ordenador, aunque

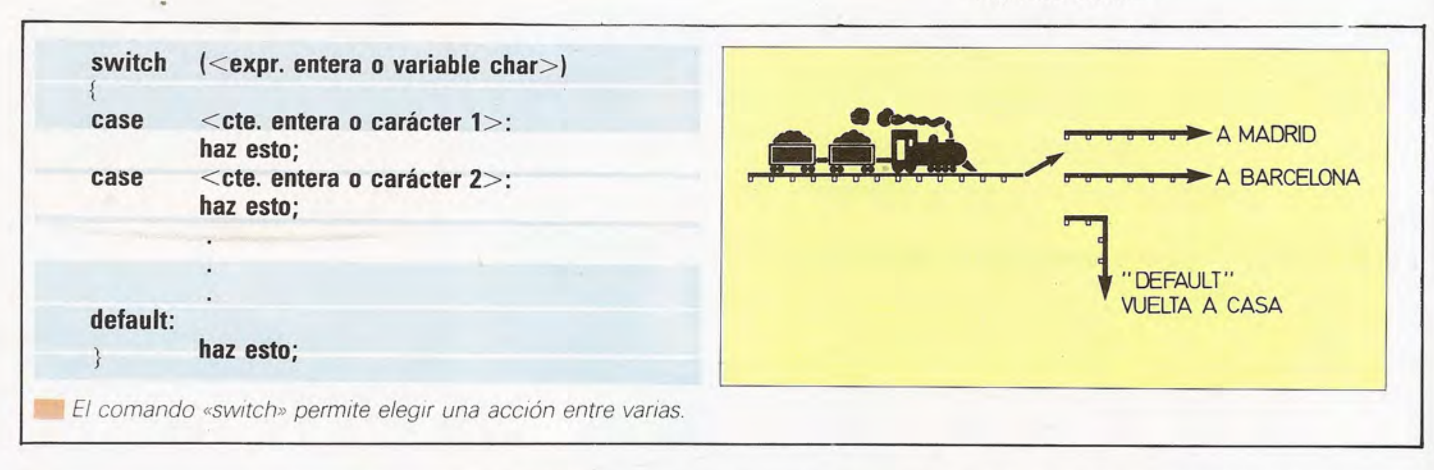
OPERADORES RELACIONALES

A == B	A ES IGUAL A B
A < B	A ES MENOR QUE B
A > B	A ES MAYOR QUE B
A >= B	A ES MAYOR O IGUAL QUE B
A <= B	A ES MENOR O IGUAL QUE B
A != B	A ES DISTINTO DE B

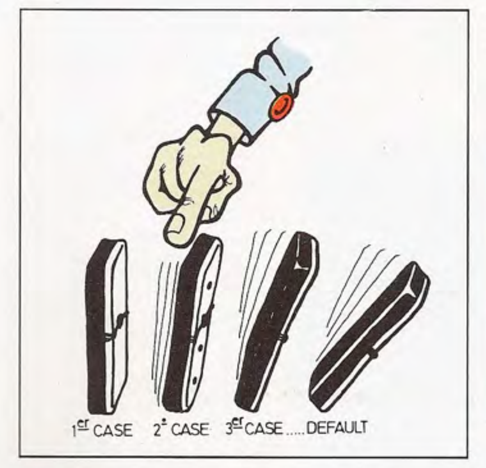
OPERADORES LOGICOS

&&	'AND' LOGICO ("Y")
II	'OR' LOGICO ("O")

Las expresiones booleanas se forman a través de operadores relacionales y lógicos.



El comando «switch» permite elegir una acción entre varias.



Un «switch» que no contenga ningún «break» ejecutará la secuencia asociada al caso correspondiente (en la figura, el segundo de ellos) y todas aquellas que la sigan hasta el final del mismo.

el nombre de "printf" está universalmente aceptado.

La salida del programa es:



con el cursor situado al principio de la siguiente línea. Este salto de línea lo produce el grupo "\n" (en algunas máquinas que no tienen el símbolo "\" se utiliza una Ñ), denominado «secuencia de escape»; como vemos, las secuencias de escape no son escritas en pantalla, sólo sirven —fundamentalmente— para controlar el cursor. Hay varios tipos, según ilustra la correspondiente figura. De no haber finalizado el argumento de "printf" con la secuencia "\n", la salida habría sido la misma pero con el cursor situado al final de la línea recién escrita.

Las secuencias de escape pueden ir en cualquier lugar del argumento; cuando se detecta una barra invertida se analiza el siguiente carácter para ver dónde se sitúa el cursor y se sigue escribiendo el resto del argumento. Así, por ejemplo:

```
printf ("primera frase\r segunda frase");
```

da como resultado:

segunda frase_

ya que \r vuelve el cursor a principio de línea y al seguir escribiendo el resto de los caracteres quedan borrados los anteriores.

Un nuevo ejemplo

En la figura aparece otro ejemplo de programa en C. Como ya se señaló, destaca la presencia de «main» al principio del listado. En el cuerpo del mismo se encuentran dos variables, "a" y "b", ambas de tipo entero (integer), que se declaran con la palabra clave "int". Esta palabra, como otras pocas que se estudiarán, es de tipo reservado; es decir, el usuario no puede utilizar variables ni funciones que se llamen "int". Además, se observa que, en C, a la vez que se declara se puede inicializar una variable,

como se hace con "a". La parte de declaración de las variables podría haberse escrito como sigue:

```
int b;
int a;
a=1;
```

O bien como:

```
int a,b;
a=1;
```

O incluso como:

```
int b,a=1;
```

siendo las cuatro formas totalmente equivalentes. En todo momento se elegirá aquella que supongamos más adecuada para la interpretación de lo que se desea expresar.

En la siguiente línea se asigna a la variable "b" el valor de la función "mi_función" cuando el parámetro de tal función vale 1.

La forma en la que las funciones utilizan los parámetros y devuelven valores es análogo a como se hace en PASCAL.

Cuando el programa llega a:

```
b=mi_función (a);
```

el valor que en ese momento tiene la variable "a" (en nuestro caso es 1) es copiado sobre la variable "i" que está en la declaración de "mi_función". En el cuerpo de "mi_función" nos encontramos una sola sentencia:

```
return (i+1);
```

```
main ()
{
   int i=2;

   switch (i)
   {
     case 1;
          printf("Estamos en el caso 1\n");
     case 2;
          printf("Estamos en el caso 2\n");
     case 3;
          printf("Estamos en el caso 3\n");
     default:
          printf("Estamos en default\n");
     }
   }
```

ESTAMOS EN EL CASO 2
ESTAMOS EN EL CASO 3
ESTAMOS EN DEFAULT

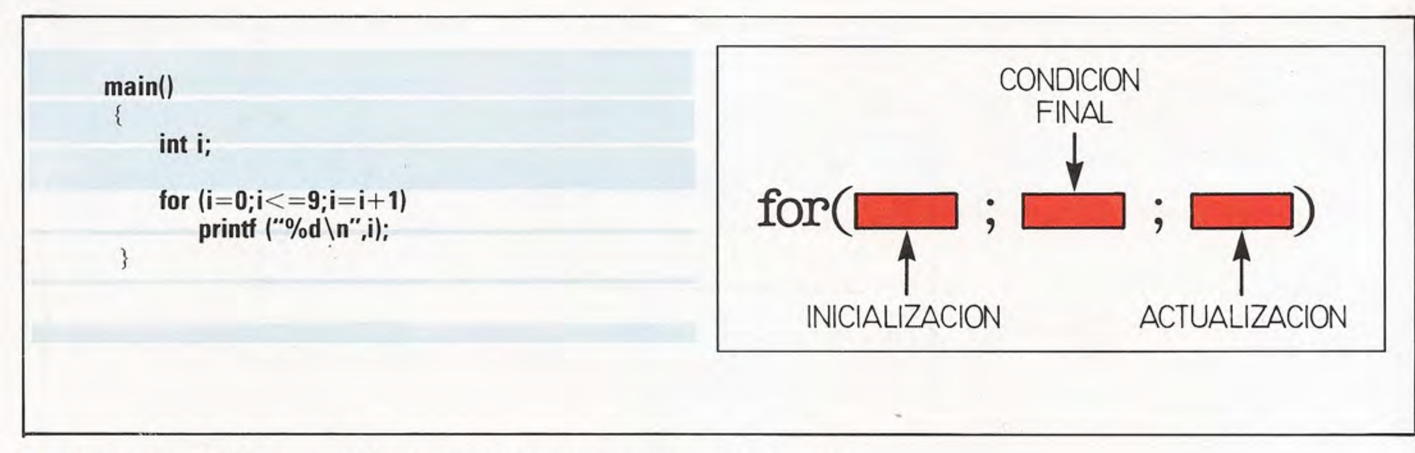
Un ejemplo del curioso comportamiento de «switch».

```
main()
{
   char c="b";
   switch (c)
   {
     case 'a':
          printf("Estamos en el caso A\n");
           break;
     case 'b':
          printf("Estamos en el caso B\n");
          break;
     case 'c':
          printf("Estamos en el caso C\n");
          break;
     default:
          printf("Estamos en default\n");
     }
   }
```

ESTAMOS EN EL CASO B

break!

En el ejemplo se observa cómo solucionar el problema planteado en el Programa 2.



```
main()
{
    int i;

    for (i=0;i<=9;i=i+1)
        printf ("%d\n",i);
}
```


La estructura FOR es, sin duda, el método básico de realizar un bucle en C.

que, como se intuye, hace que "mi_función" tome el valor "i+1" (en nuestro caso 2) y vuelva al punto de «main» desde donde fue llamada, asignando a "b" el valor 2.

En el segmento de programa correspondiente a "mi_función", y antes de comenzar el cuerpo de la misma (delimitado por {}), se encuentra la declaración:

```
int i;
```

Al igual que en PASCAL es preciso declarar los tipos de los parámetros de las funciones. La declaración equivalente en PASCAL de nuestra función sería:

```
FUNCTION mi_función (i: integer): integer;
```

En nuestro caso sólo ha sido necesario hacer la declaración del parámetro, ya que en C una declaración de función que no lleve explícito el tipo se supone que devuelve un valor entero. Si por cualquier otra causa, "mi_función" devolviera un valor real a partir de un entero, la declaración sería:

```
float mi_función (i)
int i;
```

donde "float" —también palabra clave— significa "floating point", que se corresponde con la declaración "real" de PASCAL.

Más sobre «printf»

Volviendo a «main», una vez que a "b" se le ha asignado su valor, aparece un "printf", el cual merece un comentario.

```
main()
{
    int i;

    for (i=0;i<=9;i=++i)
    {
        printf ("%d",i);
        printf ("\n");

    }
```

Aspecto de una estructura FOR cuando en su cuerpo hay que incluir más de una sentencia.

Como se indicó en un ejemplo anterior, "printf" escribirá los caracteres que van apareciendo entre las dobles comillas hasta llegar al "%". En este punto sustituirá el grupo "%d" por el valor de la primera variable que se encuentre a continuación del citado texto entre comillas, en nuestro caso el valor de "b"; luego posicionará el cursor al principio de la siguiente línea.

Nuestro programa dará por tanto la siguiente salida:

```
El resultado de mi función es 2
_
```

La forma general de "printf" se observa en la figura adjunta. El "string de formato" irá entre dobles comillas y será lo que aparezca en la pantalla, además de los valores de los argumentos en los lugares indicados por los símbolos "%"; estos símbolos van siempre seguidos por una letra (juntos forman un «especificador de campo») que indica el tipo de argumento que se va a escribir. Las distintas especificaciones se dan en la correspondiente figura.

Los tipos de datos

En C existen cuatro tipos básicos de datos. De ellos sólo "int" y "float" son realmente básicos, los otros surgen como derivaciones o ampliaciones. Con estos cuatro tipos, y de manera parecida a como se hace en PASCAL, pueden formarse estructuras más complejas, ta-

les como arrays (equivalente a los "strings" de C) o records ("struct" en C).

Para declarar variables se anteceden éstas con la palabra clave correspondiente a su tipo, seguida por los nombres de las variables. Las palabras clave son:

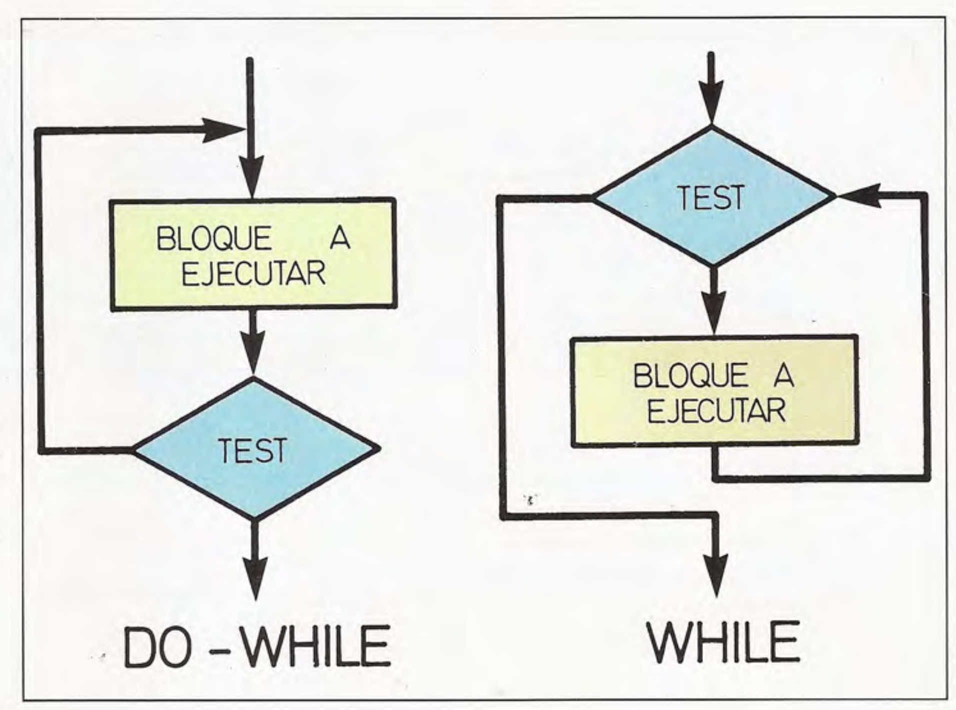
int: entero
char: carácter
float: números reales
double: números reales en doble precisión

Un valor real en doble precisión es un valor tipo "float" (real) especificado con mayor exactitud, aunque no tiene por qué ser el doble de preciso.

Los caracteres se representan entre comillas simples, por ejemplo:

```
char primera, última;
primera='A';
última='z';
```

La diferencia entre lo que significa 'A' y "A" se evidenciará al tratar de los strings.



La diferencia entre un «do-while» y un «while» la establece el distinto lugar en el que se realiza la comprobación de la certeza o falsedad de la condición del bucle.

```
main()
 {
     int i, n;

     for (i=2; n<=1000 ; ++n)
     {
        for (i=2; i<n ; ++i)   /* esto es el */
            if (n%i==0)        /*   bucle    */
              break;           /*  interno   */
        if (i==n)
          printf ("%d\n",n);
     }
```

¡Por fin un programa útil en C!.

Tipos de almacenamiento

En C una variable tiene asignado, aparte de un tipo de dato, un tipo de almacenamiento. Para obtener una idea al respecto, cabe pensar en la diferencia que existe entre variables locales y globales de PASCAL.

Variables automáticas

Suponga que se modifica "mi función" tal como muestra la correspondiente figura. En este caso, "cte" es una variable automática. Se caracteriza porque sólo puede ser referenciada dentro de "mi_función", esto es: su utilización fuera de tal ámbito daría un error. Las variables automáticas tienen existencia mientras se está en el bloque donde son declaradas, nada más. De esta forma sólo ocupan memoria cuando son necesarias, lo cual es a menudo muy ventajoso. La forma ortodoxa de declarar "cte" es la que sigue:

```
auto int cte;
```

Si bien, la palabra clave "auto" se suele omitir por la frecuencia con la que aparecen este tipo de variables. El compilador supone que toda variable sin especificar el tipo de almacenamiento es automática.

Variables de registro

Su utilización y características son análogas a las del tipo automático. En realidad se trata de indicar al compilador que tal variable será utilizada con mucha frecuencia y conviene tenerla almacenada en un lugar al que se pueda acceder con facilidad, como es un registro de la CPU, en vez de tenerla en una posición de memoria. La forma de declararlas es:

```
register int cte;
```

Variables estáticas

Este tipo de variables, al contrario que los tipos anteriores, no dejan de existir al abandonar el ámbito donde son declaradas. Al igual que los tipos anteriores sólo se pueden referenciar en el ám-



Un «recurso abstracto» es la barrera que separa y une a la vez dos campos distintos.

bito de declaración, pero el valor que tenían al abandonar el bloque lo volveremos a encontrar al entrar de nuevo en él. La figura adjunta incluye un ejemplo clarificador.

Variables externas

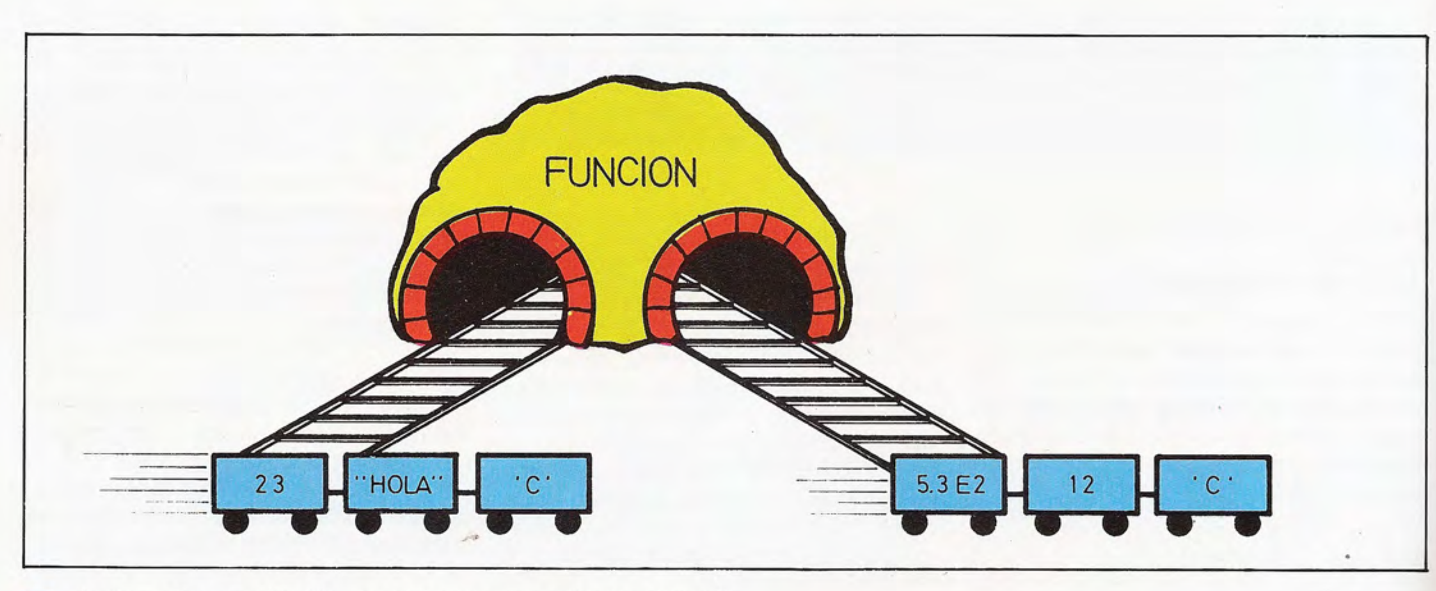
Una variable externa es accesible desde cualquier función del programa. Se declaran al principio del listado. Por ejemplo, en la figura correspondiente: «x» es global a «main» y a cualquier otra función que hubiera. Estrictamente, al principio de «main» debería aparecer la declaración:

```
extern int x;
```

El uso de este tipo de declaraciones es optativo, aunque es conveniente incluirlas si el programa es muy complicado.

La estructura if-then-else

Antes de entrar en los aspectos más significativos del lenguaje C, conviene conocer las estructuras que permiten dirigir la ejecución del programa hacia diversos puntos (selección) y las que, en



La llamada a una función supone entrar en un túnel que puede resultar sin salida, en el caso de caer en un bucle infinito.

ocasiones, obligan a apretar apresuradamente el botón reset del ordenador (bucles o iteraciones).

La primera estructura que veremos será la IF/THEN/ELSE, que, en C, tiene el siguiente aspecto:

```
if      (esta condición es cierta)
        ejecutar esto;
else
        ejecutar esto otro;
```

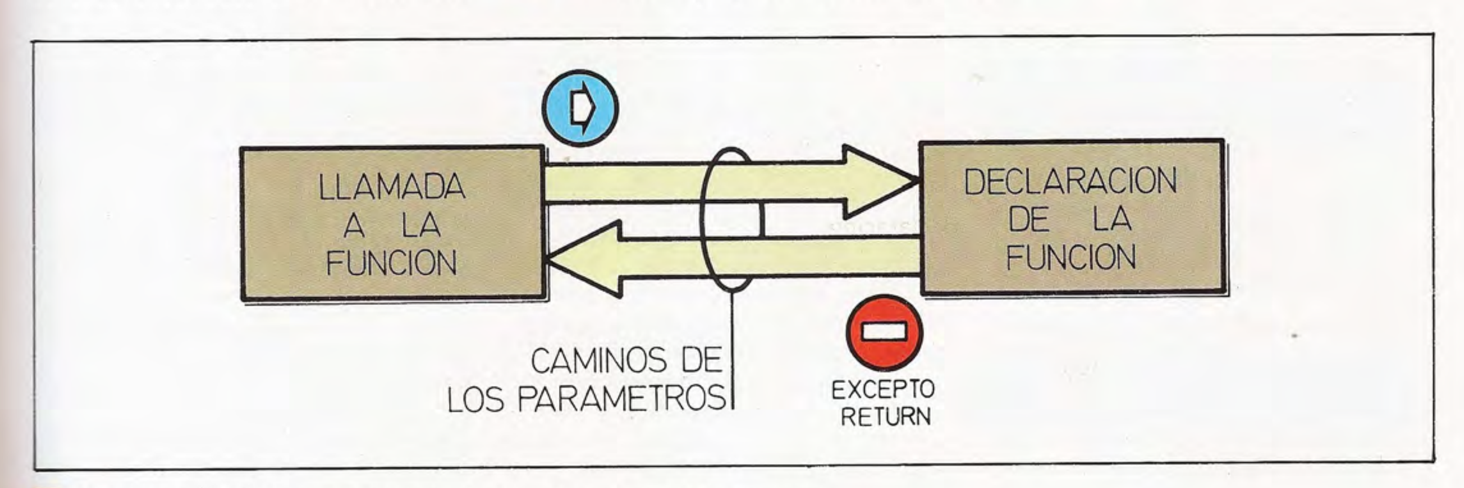
Se observa que no hay "then". Al contrario que en otros lenguajes, "then" no es palabra reservada en C, por lo que es posible declarar variables con dicho nombre.

Las selecciones "ejecutar esto" y "ejecutar esto otro" pueden estar formadas por una única sentencia o por un grupo de ellas encerradas entre llaves. Esto mismo es lo que se hacía en PASCAL sin más que sustituir las llaves por BEGIN-ENDs.

```
#define E 2.718281828
 main()
 {
      double número,su_cuadrado;
      double al_cuadrado();

      número= E;
      su_cuadrado= al_cuadrado
      (número);
      printf("%f\n",su_cuadrado);
 }
 double al_cuadrado (x)
 double x;
 {
      return (x*x);
 }
```

¡Atención con las declaraciones de funciones! El programa constituye un ejemplo correcto de declaración.



En el lenguaje C el camino hacia las funciones es de una sola dirección.

La condición que está entre paréntesis a continuación del "if" se forma a base de «operadores relacionales» y «operadores lógicos»; ambos tipos reflejados en el cuadro adjunto.

Los operadores relacionales son más o menos familiares para el aficionado, excepto el último, que en otros lenguajes se representa con el símbolo "<>". Al examinar la lista de operadores lógicos nos puede surgir una pregunta: ¿por qué && y no simplemente &? La razón es que los operadores simples como & o | están reservados para operaciones con bits, de las que se hablará sucintamente en otro momento. En cuanto al orden de evaluación, los operadores relacionales se evalúan siempre antes que los lógicos, por lo que:

```
a<b && b==c
```

es totalmente equivalente a:

```
(a<b) && (b==c)
```

Dentro de los lógicos tienen mayor preferencia las expresiones relacionadas por && que las que lo están por ||.

La estructura swicht

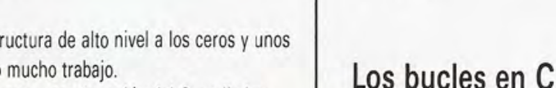
No siempre son tan sencillas las cosas para que existan sólo dos alternativas a una pregunta. Para estos casos más complicados se dispone de la estructura "swicht" (equivalente al CASE del PASCAL). Su aspecto queda reflejado en la figura adjunta.

Al llegar a un "swicht" se evalúa la expresión que está entre paréntesis y se exploran secuencialmente los "case" hasta encontrar uno cuya constante coincida con el resultado obtenido al evaluar la expresión. Si no hay ninguna

coincidencia, se ejecutará la acción que sigue a "default". Hay que resaltar que, al contrario que en el CASE de PASCAL, una vez que ha ocurrido una coincidencia con algún "case", se ejecutará la acción asociada a él y el resto de las opciones que lo siguen, hasta llegar al final de la estructura, señalado por la .

Esta forma de operar del "swicht" puede sorprender hasta el punto de dudar de su utilidad. Para obviar este problema se hace uso de la sentencia "break", la cual obliga a pasar el control justo a continuación del bloque delimitado por { . Al incluirla en un "swicht" se obligará inmediatamente a pasar el control al punto donde está el " } " que indica el final del "swicht", tal y como se observa en la correspondiente figura.

De código fuente a código ejecutable



Pasar de una estructura de alto nivel a los ceros y unos cuesta a menudo mucho trabajo.
Hay veces en las que a continuación del Compilador actúa el "OPTIMIZADOR", siguiendo el resto de los pasos como muestra la figura. El optimizador proporciona un código ensamblador más reducido, lo que irá en favor del tiempo de ejecución de nuestro programa.

Los bucles en C

El lenguaje C brinda tres tipos de estructuras para realizar bucles, que tienen sus equivalentes en los FOR-DO, REPEAT-UNTIL y WHILE-DO del PASCAL.

- *El bucle "for"*

En la figura adjunta se incluye un programa cuya acción es escribir en columna los diez primeros números naturales. Como se observa, a continuación de la palabra clave "for" se encuentra un paréntesis con tres parámetros que controlan el bucle. El primero de ellos es el de inicialización y sólo se ejecuta en la primera entrada al mismo. El central expresa la condición para la cual el bucle seguirá ejecutándose. El último proporciona la actualización de la variable de control del bucle.

En el ejemplo, tal actualización es un simple incremento de una variable. Para realizar incrementos de variables en C podemos recurrir a una de las siguientes expresiones:

```
i=i+1;
i++;
++i;
i+=1;
```

Mientras que para decrementar una variable se pueden utilizar las mismas expresiones cambiando "+" por "–".

La diferencia entre la segunda y la tercera de las formas indicadas reside en que, en la segunda, primero se utiliza la variable y luego se incrementa, mientras que las cosas ocurren en orden contrario en la tercera. Aunque en el ejemplo propuesto ambas formas darán el mismo resultado, se verán otros ejemplos en los que el elegir una u otra

forma es muy importante. La cuarta forma puede ser útil cuando haya que incrementar una variable con un nombre muy largo, o una expresión compleja, ya que sólo hay que escribirla una vez. Como es habitual, si el cuerpo del bucle está formado por más de una sentencia hay que encerrar a éstas entre { } (ver figura).

- *El bucle "while"*

Su forma es:

```
while (esto sea cierto)
        hacer esto;
```

Nuevamente, el cuerpo del "while" debe ir encerrado entre llaves si está formado por formado por más de una sentencia.

- *El bucle "do-while"*

Su equivalente es el REPEAT-UNTIL del PASCAL y presenta el siguiente aspecto:

```
do
     esto;
while (esta condición sea cierta);
```

La diferencia entre este y el anterior reside en que el bucle "do-while" siempre es ejecutado al menos una vez (hay que observar dónde se produce el «test» para decidir si se ejecuta de nuevo o no), mientras que el "while" puede no ser ejecutado ni una sola vez.

Las condiciones que tanto en uno como en otro caso aparecen a continuación de la palabra "while" se forman de manera análoga a como se hace con las de las estructura "if-then-else".

Afianzando conceptos

A modo de resumen de las estructuras descritas cabe proponer el ejemplo práctico de la figura adjunta.

Como se vio al principio del capítulo, el lenguaje C está todavía en su más tierna infancia y hay muchos aspectos del mismo que son claramente mejorables. Tal sería el caso de los operadores lógicos, reduciendo el símbolo doble a uno simple, por ejemplo. Además, se ha visto que el "swicht" de C es mucho menos potente que su equivalente en el PASCAL. Como sucede en otros campos, cuanto más se aligere el trabajo de la máquina, más se tendrá que cargar de trabajo el hombre; con el C se pretende tener un lenguaje compacto, cuyo compilador sea fácil de diseñar y ocupe poca memoria. Haciendo que sea el programador quien se ocupe de ciertas tareas que debieran estar a cargo del compilador (caso del "swicht"), se puede conseguir el objetivo arriba citado.

Volviendo al ejemplo, se trata de un programa que calcula los números primos entre 2 y 1000. Tal vez sea conveniente recordar que un número primo es aquél que sólo es divisible por sí mismo y por la unidad; y esta será la propiedad que se utilizará en el programa. Se parte de dos bucles, uno que generará candidatos a primos y otro que se dedicará a comprobar la citada propiedad. Observando el bucle interno de la figura, se deduce que si se ha detectado que «n» (candidato) es divisible por un número, se abandona dicho bucle para comprobar si el divisor encontrado es el propio «n», en cuyo caso será un número primo; si no es así, se retorna al bucle exterior para asignar a «n» un nuevo candidato.

Salta a la vista que el uso de "break" no se restringe al ámbito de los "swicht". Además, se ha introducido el operador "%", que calcula el resto de la división entre dos números. Si este resto es cero, el primer número es divisible por el segundo.

```
main()
{
     int x=6;

     printf("%d\n",factorial(x));
}
factorial (n)
int n;
{
     if (n==1)
          return (1);
     else
               return (n*factorial(n_1));
}
```

Un ejemplo de función recursiva.

El siempre presente ";"

Los conocedores del Lenguaje Pascal están acostumbrados a «dejar caer» un ";" al final de casi todas las sentencias.
En C también hay que utilizar el ";" muy a menudo, aunque su significado respecto a Pascal varía sutilmente.
En Pascar, un ";" se utiliza como separador de sentencias, mientras que en C se usa como terminador de sentencias. Por esta razón, nunca va un ";" antes de un ELSE en Pascal, ya que entonces el compilador tomaría a ELSE como principio de una nueva sentencia, y esto no está de acuerdo con la sitaxis del Pascal. En C, sin embargo, si aparecerán ";" antes de los "else" ya que, por su definición sintáctica, siempre ha de seguir una sentencia a un "if", y una sentencia debe ser terminada por un ";", por lo que si a continuación hay un "else", éste se verá precedido del dichoso símbolo.

Algo más sobre funciones: el «recurso abstracto»

Una función puede compararse con una «caja negra», en la que se introducen ciertos valores, y la cual devuelve otros valores de salida como resultado de procesar los primeros. Siempre que se crea una función, aparte de estar

Lista de palabras clave del lenguaje «C»			
auto	double	if	static
break	else	int	struct
case	entry	long	swicht
char	extern	register	typedef
continue	for	return	union
default	float	short	unsigned
do	goto	sizeof	while

Especificadores de campo para «printf»	
Valores enteros:	%d escribir argumento como valor decimal
	%o escribir argumento como valor octal
	%x escribir argumento como valor hexadecimal
Valores reales:	%f escribir el argumento en la forma ddd.ddd
	%e escribir el argumento en la forma d.ddd+dd
caracteres:	%c escribir un carácter
cadenas:	%s escribir un string de caracteres

contribuyendo a dar un carácter modular al programa, se está haciendo referencia a un «recurso abstracto»: un recurso (un medio, una facilidad) que no posee la máquina en sí y que el usuario le entrega a partir de ese momento.

Veamos un ejemplo:

En C no existe operador de potenciación, así que si se quieren realizar elevaciones al cuadrado, habrá que dotar al ordenador de un recurso (abstracto, porque en realidad no lo tiene) tal como éste:

```
double al_cuadrado (x)
double x;
{
        return (x*x);
}
```

Aún a pesar de sus connotaciones filosóficas, no hay que perder de vista que los recursos abstractos son pilares fundamentales sobre los que descansa la metodología de la programación estructurada.

Unas líneas más arriba se ha declarado una función que devuelve un valor en doble precisión. En una de las figuras aparece un programa que la utiliza. Como se observa, si se utiliza en un ámbito («main» en nuestro caso) una función que no devuelve un entero, es preciso declarar en tal ámbito el tipo de la función:

```
double al_cuadrado ();
```

Esto es necesario porque, como ya se comentó, todo identificador del que no se especifica el tipo se supone entero. Si no se hubiera declarado "al_cuadrado" en «main», el compilador supondría que iba a devolver un entero, lo cual está en desacuerdo con la posterior declaración.

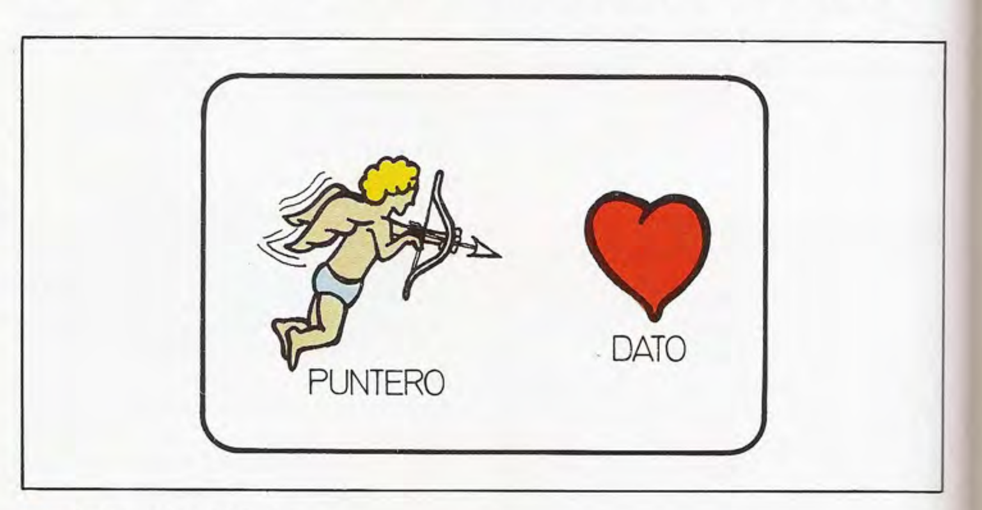
Otro punto importante: en PASCAL, los parámetros de un procedimiento podían pasarse «por valor» o «por referencia» (estos últimos también llamados "VAR". Los parámetros VAR establecen una doble vía de comunicación entre la parte «llamadora» y la «llamada», ya que que el parámetro actual es modificado al finalizar la ejecución del procedimeinto. En C todos los parámetros pasados lo son por valor, es decir, el único medio que hasta ahora conocemos de comunicación en C entre una función y su «llamador» es a través de "return". Como luego se verá, la forma de «imitar» a un parámetro VAR es a través de los punteros.

Para finalizar, hay que mencionar que al igual que en PASCAL, una función puede llamarse a sí misma recursivamente. La recursividad es una propiedad muy importante y que sirve para muchísimas más cosas que para calcular factoriales.

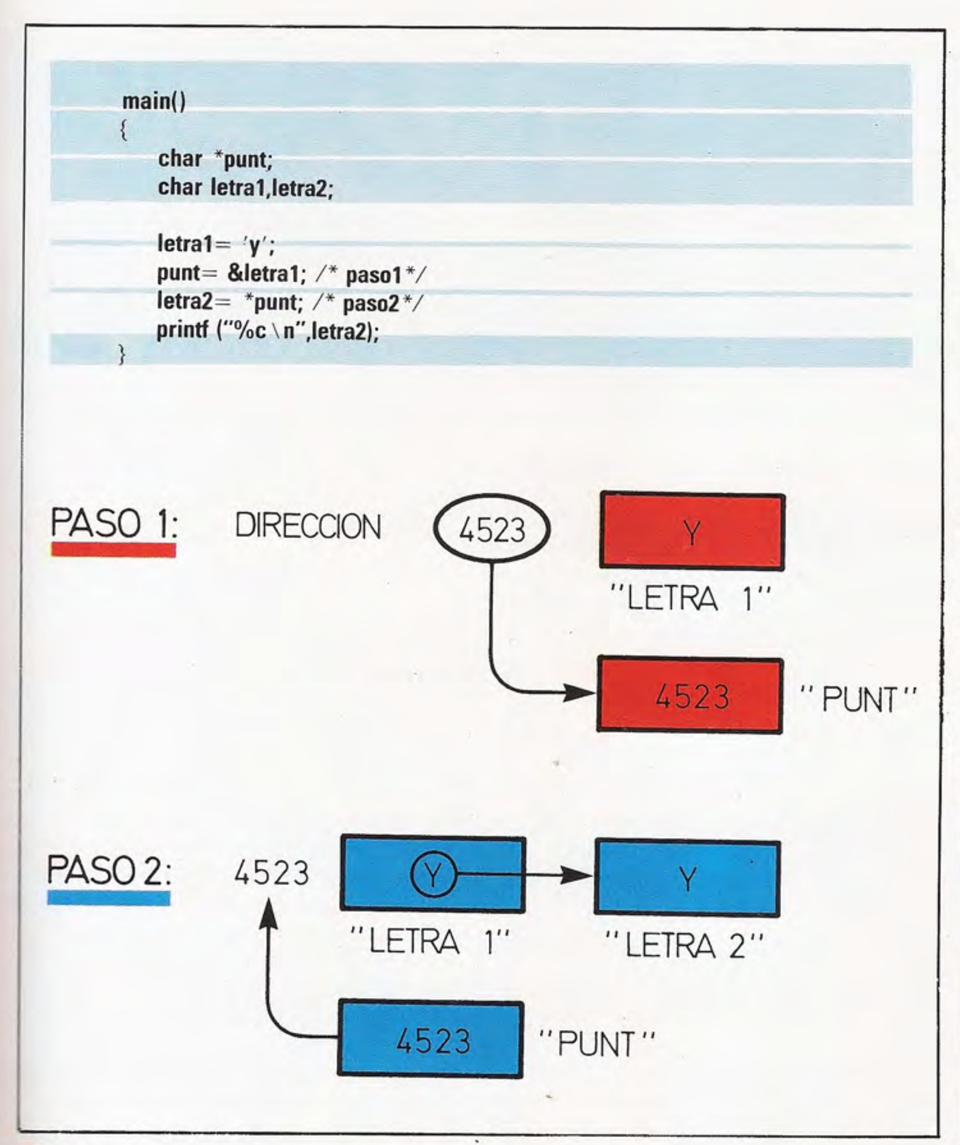
Los punteros del C

C es un lenguaje que hace un uso intensivo de punteros. Hay ocasiones en las que la única manera de expresar un cálculo es a través de punteros. Otras veces el uso de punteros reduce el tamaño del código objeto de los programas, aumentando su velocidad de ejecución. No obstante, el uso indiscriminado de punteros puede conducir a la creación de monstruos (programas) irreconocibles incluso para su propia madre (el programador). Hay que procurar, por lo tanto, ser cuidadosos con su uso.

Conviene recordar en este punto que, por muy inspirados que sean los nombres otorgados a las variables y por muy estructurado que resulte el programa,



Los punteros son un recurso muy frecuente en la programación en lenguaje C.



Punteros: viaje al país de los trabalenguas.

tras la compilación sólo quedarán direcciones y datos. Es bueno, por tanto, saber qué pasa con las direcciones y los datos.

Un puntero es una variable que no contiene un dato, sino que se utiliza para apuntar a una variable; es decir, contiene la dirección de memoria en la que se encuentra la variable. Puede accederse al dato contenido en la variable directamente, a través de su nombre, o indirectamente, a través de un puntero que la señale.

Con un puntero se pueden hacer básicamente dos cosas:

1.º Almacenar en el puntero la dirección de una variable (operador &).
2.º Acceder al valor contenido en la dirección señalada (operador *).

Para comenzar, tomemos la siguiente declaración de puntero:

```
char *punt;
```

Ello significa que "punt" señalará hacia una posición de memoria que contiene un carácter o de otra manera, que "*punt" contendrá un carácter. Sustituyendo "char" por otros declaradores de tipo se tendrán punteros a distintos tipos. Ahora, podemos declarar e inicializar algunas variables:

```
char letra1,letra2;
letra1='y';
```

Al hacer:

```
punt=&letra1;
```

la variable "punt" está apuntando hacia "letra1", y al hacer:

```
letra2=*punt;
```

"letra2" contiene el dato al que apuntaba "punt". Todos estos pasos se ven reflejados en la correspondiente figura.

El resultado de todo este ir y venir es análogo al que se hubiera obtenido haciendo:

```
letra2=letra1;
```

una vez que se asignó a "letra1" el valor 'y'.

Las constantes simbólicas

C permite el uso de las llamadas «constantes simbólicas» para que los listados sean inteligibles y los programas fácilmente modificables.
Al igual que en la sección CONST de Pascal, una constante simbólica permite, con la sola modificación de una línea, variar la acción del programa para que pueda ser utilizado en distintas aplicaciones.
Las constantes simbólicas se pueden colocar en cualquier parte del programa, aunque suelen situarse al principio del mismo. El ejemplo universal de constante simbólica es el número PI.
En Pascal se incluiría:

```
CONST PI-3.14159;
```

En C, una constante se especifica de la siguiente forma:

```
#define PI 3.14159
```

(cuidado con poner el ;)

La misión del preprocesador, del que se habló en el capítulo precedente, consiste en analizar el programa para encontrar la palabra PI y sustituirla por todo aquello que sigue al segundo espacio en blanco de la declaración «#define». De ahí que no haya que incluir el signo «;» en tal declaración.

```
main ()
{
    int a=5,b=10;

    printf("%d %d \n",a,b);
    cambia (&a,&b);
    printf("%d %d \n",a,b);
}
cambia (px,py)
int *px,*py
{
    int temporal;

    temporal= *px;
    *px= *py;
    *py= temporal;
}
```

```
5 10
10 5
```

Un parámetro tipo VAR se pasa a través de la dirección de la variable, no de su valor concreto. La pantalla adjunta muestra el resultado de la ejecución del programa adjunto.

Argumentos de funciones y punteros

En este punto, se está ya en disposición de tratar el problema del paso de argumentos por referencia. Se puede plantear el problema de requerir una función que devuelva dos valores; esto no puede lograrse tal como se venía haciendo hasta ahora, a través de "return". Suponga una función que debe intercambiar los valores de dos variables enteras. Si ya se tienen las variables declaradas y asignadas, no bastaría con escribir:

```
cambia (a,b);
```

en donde "cambia" ha sido definido como sigue:

```
cambia (x,y)
int x,y;
{
        int temporal;
        temporal=x;
        x=y;
        y=temporal;
}
```

ya que como en C los parámetros son pasados por valor, a "x" e "y" les son asignados los valores de "a" y "b", pero al finalizar "cambia" las variables originales permanecen inalteradas.

Lo cierto y lo falso en el «C»

Una constatación inmediata en que en el lenguaje C no existe el tipo booleano en sí. Esto se debe a que para el C todo aquello que sea distinto de cero es cierto. Suponga el siguiente segmento de programa:

```
if (es_cierto)
    función1();
else
    función2();
```

donde la variable «es_cierto» se ha declarado de tipo «int». Si al llegar al «if» su valor es cero, se ejecutará la función 2 («función2». Por el contrario, si su valor es cualquier otro excepto el mencionado, se ejecutará «función1». Otra forma de tratar variables booleanas es a través de un par de sentencias «#define«;

```
#define TURE 1
#define FALSE 0
```

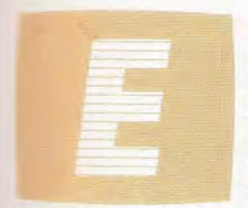
Es posible definir «TRUE» con cualquier valor distinto de cero, aunque se suele tomar el valor «1» por simplicidad.

La solución consiste en pasar las direcciones de "a" y "b" en vez de sus valores, realizando en "cambia" un intercambio de contenido de direcciones tal como se observa en la ilustración. Al llamar a "cambia" desde «main» se asignan a "px" y a "py" las direcciones de "a" y "b", no sus valores como se venía haciendo hasta ahora. Una vez en el cuerpo de "cambia", cada vez que se escribe "*px" o "*py" se hace referencia a los contenidos de las direcciones a las que apuntan "px" y "py", que son las de "a" y "b". Aunque resulte complicado a primera vista, se recomienda al lector que trabaje sobre este ejemplo y se dé cuenta de que realmente se consigue el efecto deseado.

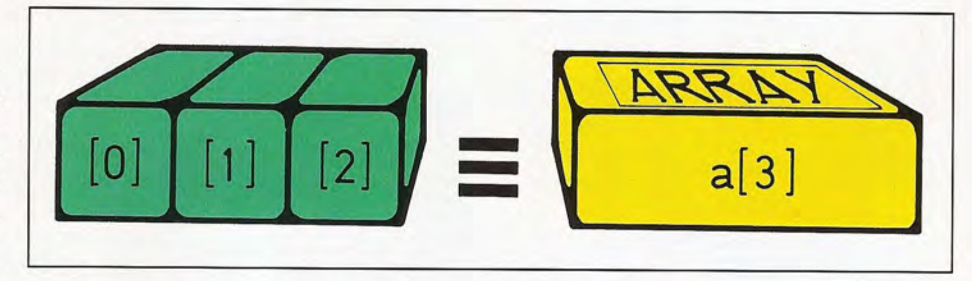
Cabe hacer algún comentario adicional sobre las funciones en C. En primer lugar, las funciones no tienen por qué llevar obligatoriamente un "return"; puede pensarse en lo innecesario de un "return" en una función que simplemente ejecuta un bucle para producir un retardo. Por lo demás, como consecuencia de la no existencia del "return", el tipo de la función es irrelevante, ya que no es asignado el resultado a ningún valor, como ocurre en la llamada a "cambia" que se produce en «main».

C (y 2)

Estructuras de datos: arrays y registros.



El objetivo del presente capítulo lo constituye el estudio de las estructuras de datos del lenguaje C. Como ya se ha comentado, con el «C» se pretendía obtener un lenguaje compacto y fácilmente implementable en casi cualquier ordenador. Por esta razón, las estructuras de datos de este lenguaje destacan por su sencillez, aunque suficientes para la mayoría de las aplicaciones. Antes de comenzar con su estudio es conveniente ampliar conocimientos relativos a punteros.



En C, un array declarado como «int a[3];» no contiene el elemento a[3]. Dicho array contendrá los elementos a[O], a[1] y a[2]. La razón se comprenderá plenamente al estudiar la relación entre punteros y arrays. De momento, basta con considerar que los arrays comienzan con el elemento cero, en lugar de con el uno.

Punteros y aritmética

Los punteros son piezas clave en el desarrollo de casi cualquier programa. Por esta razón, las operaciones con y entre punteros no se reducen a lo ya descrito en el capítulo precedente, sino que además está permitido:

— Incrementar o decrementar un puntero.

— Sumar o restar un entero a un puntero.

— Comparar y substraer dos punteros, en el supuesto de que ambos señalen al mismo tipo de objeto.

Cuando se incrementa o decrementa un puntero, el lenguaje C tiene en consideración el hecho de que no todo los tipos de datos ocupan igual número de posiciones de memoria, tal como se observa en el cuadro adjunto.

Cabe recordar que toda la información que contenga el ordenador ha de quedar reducida a ristras de ceros y de unos, reunidos en grupos de ocho. Suponga una sentencia como la que sigue:

```
char *caractpunt;
```

la cual declara a "caractpunt" como puntero a un carácter. Haciendo referencia a la zona de memoria representada en el cuadro «Los datos en memoria", puede lograrse que "caractpunt» apunte a la dirección del primer carácter de la misma a través del operador "&", tal como se expuso en el capítulo precedente.

Si ahora se ejecuta la orden:

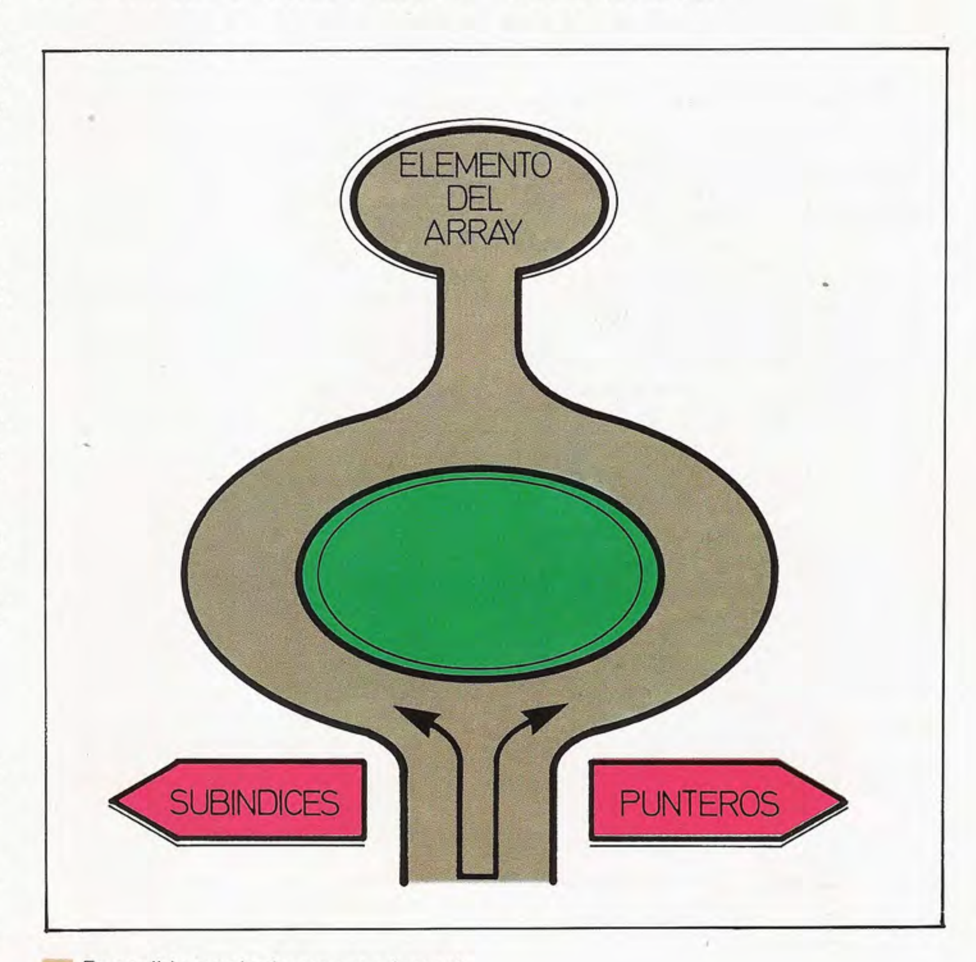
```
++caractpunt;
```

"caractpunt" se incrementa en una unidad, y pasa a apuntar al siguiente carácter almacenado.

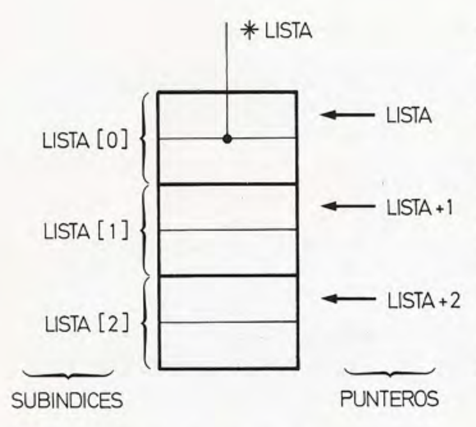
De forma análoga, puede lograrse que un puntero señale al primer entero de la zona de memoria (posición 1125) declarando:

```
int *enteropunt;
```

esta vez, al hacer:



Es posible manipular arrays a base de subíndices, como es habitual en otros lenguajes, o a base de punteros y su aritmética.



El nombre de un array es, de hecho, un puntero orientado al elemento cero del mismo.

++enteropunt;

su valor no se incrementará en una unidad, sino en dos, para apuntar al siguiente entero.

Análogamente, si aparece una sentencia en la que hay que incrementar el valor de un puntero para señalar hacia un dato de tipo real, el incremento debe ser el necesario para «saltar» la zona de memoria ocupada por el dato y quedar apuntado al inmediato posterior.

Todo lo expuesto para las operaciones de incrementar o decrementar es aplicable al caso de una suma o diferencia de un puntero y un número entero. Si se tiene un puntero que señala a un valor entero y se ejecuta:

enteropunt=enteropunt+3;

el verdadero incremento que sufrirá ''enteropunt'' será de seis unidades y su efecto será el de saltar tres enteros consecutivos de la zona de datos sobre la que actúe.

En los próximos apartados se analizarán ejemplos concretos de la aritmética de punteros.

La estructura array

Si hay una forma sencilla de agrupar una serie de datos es por medio de un array. La manipulación de los arrays en C es muy parecida a la habitual en los restantes lenguajes de alto nivel; sus peculiaridades aparecen al relacionar los arrays con los punteros.

Veamos en primera instancia cómo se declara un array. En el lenguaje C no existe una palabra clave al efecto, sino que, por ejemplo:

int a[6];

es la declaración de un array de nombre «a» que contendrá seis elementos enteros. De igual forma:

float num[100];

declara un array de cien elementos que serán números expresados en punto flotante. Para referenciar un elemento concreto del array se hace seguir al nombre de un subíndice; este subíndice puede ser una constante, una variable, o una expresión cuya evaluación sea un número entero, encerrado entre corchetes. Así, volviendo a la declaración de «a», para asignar el valor 10 al tercer elemento se procederá como sigue:

a[2]=10;

¿Hemos dicho tercer elemento? Más bien parece el segundo. En la figura adjunta aparece la justificación de esta aparente incongruencia.

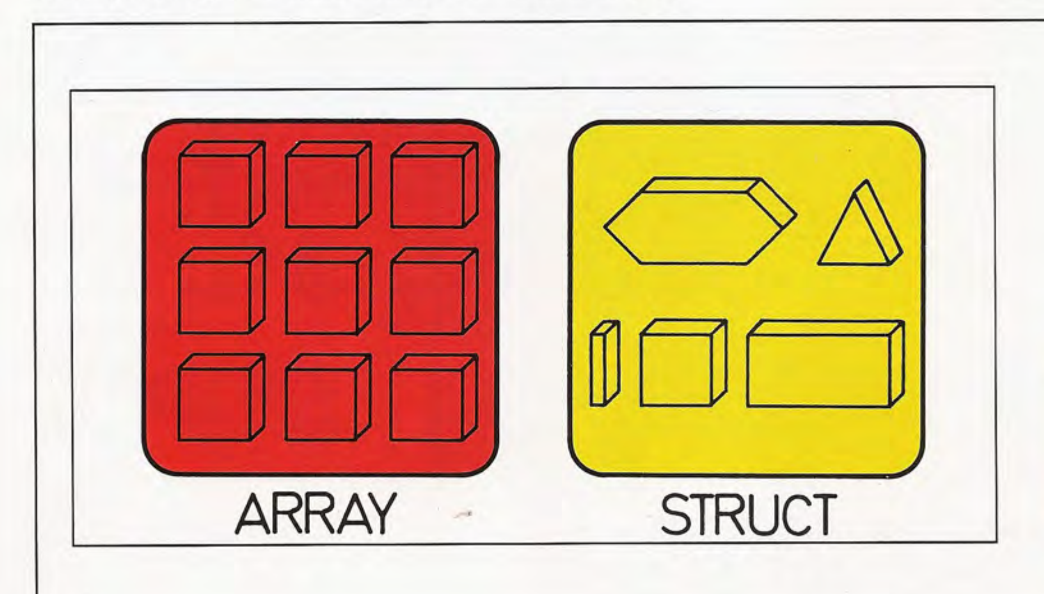
Representación interna

Suponga un segmento de programa como el siguiente:

```
int lista[3];

lista[0]=2;
lista[1]=5
lista[2]=-3;
```

El compilador de «C» al reconocer la declaración de ''lista'' en la primera línea, reserva tres espacios de memoria consecutivos a partir de una dirección



La diferencia entre arrays y «struct» la establecen los tipos de elementos que uno y otro pueden contener. Los arrays sólo puede incluir datos de un mismo tipo, mientras que las «struct» carecen de esta restricción.

de comienzo. Según esto, el acceso a un array puede interpretarse como tomar la dirección de comienzo y sumarle el desplazamiento necesario para acceder al elemento designado: para el elemento cero el desplazamiento es cero, para el primero es uno, y así sucesivamente.

¿No tiene este método un gran paralelismo con la técnica de los punteros? De hecho, en el C, el propio nombre de un array ("lista" en nuestro caso) constituye un puntero que señala hacia el elemento cero del mismo.

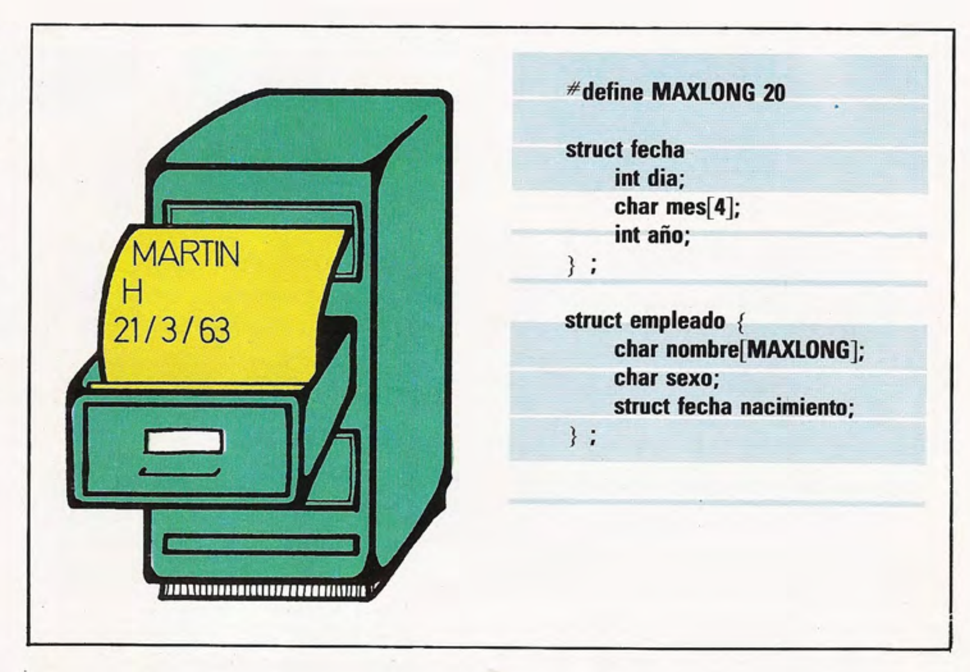
Por esta razón, y considerando lo dicho acerca de la aritmética de punteros, las expresiones:

lista[1]

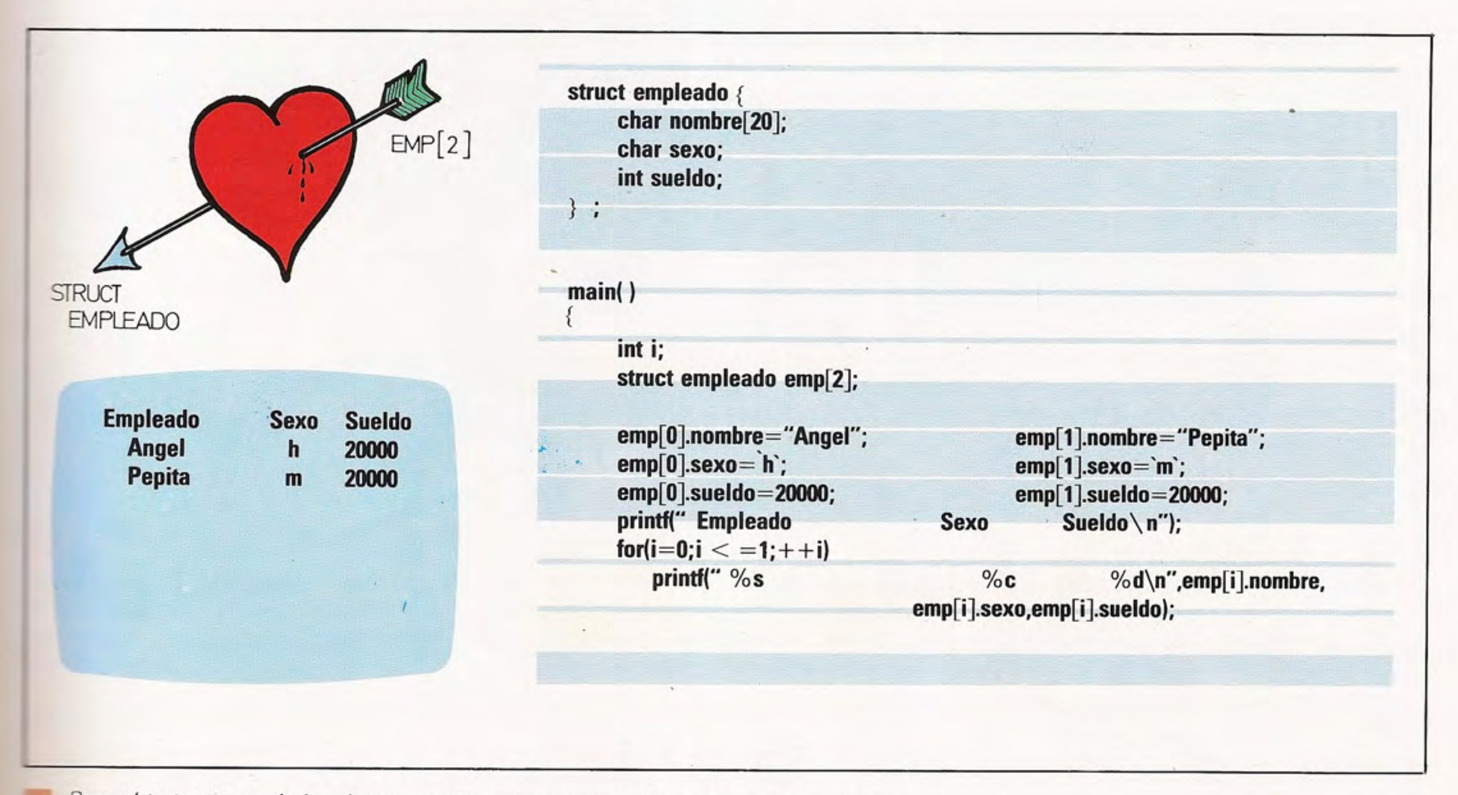
*(lista+1) y

son absolutamente equivalentes.

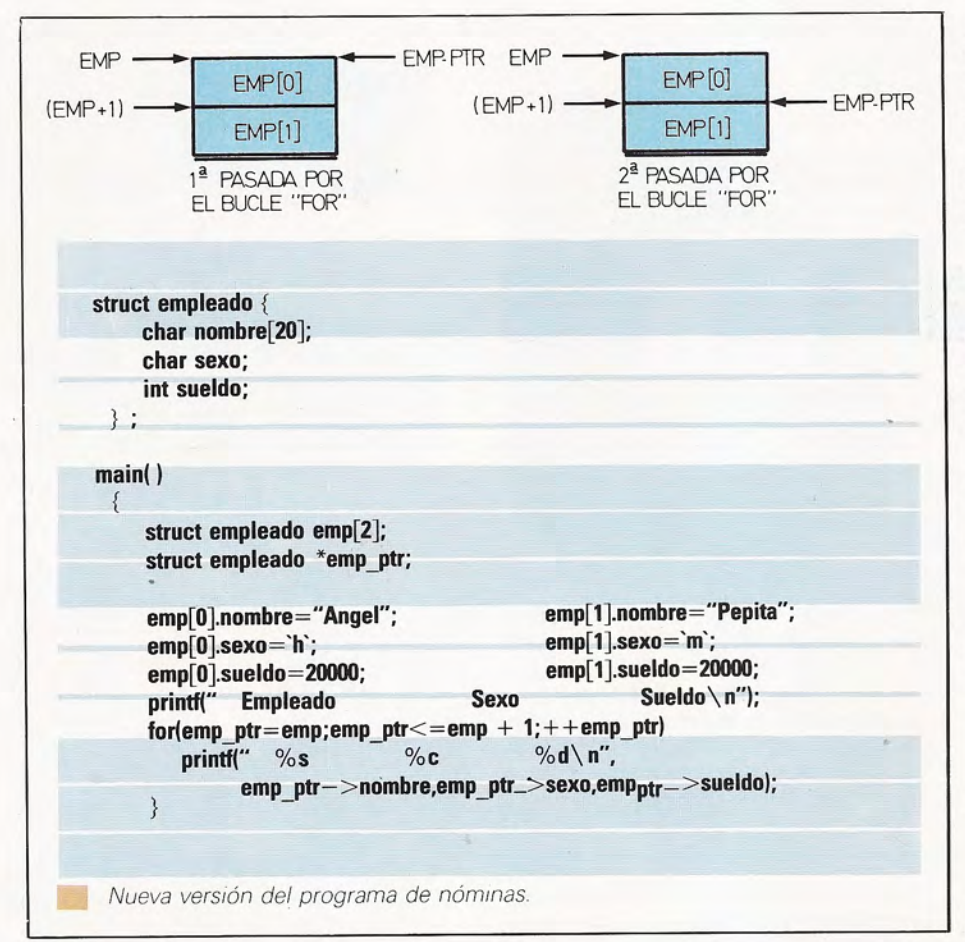
Normalmente, para referenciar al segundo elemento del array se utilizaría la expresión "lista[1]", pero como "lista" es un puntero al elemento "lista[0]", que es un entero, la expresión "lista+1" apuntará dos posiciones más adelante



El programa adjunto muestra un ejemplo de anidación de «struct». Su contenido coincide con las primeras líneas de un programa para gestionar la nómina de una empresa.



Para el tratamiento de la nómina pueden almacenarse datos sobre los empleados (suponemos que nuestra empresa tiene tan sólo dos) en un array; cada elemento de este último será una «struct» con los datos de la persona correspondiente.




```
struct empleado {
      char nombre[20];
      char sexo;
      int sueldo;
   } ;

main( )
   {
      struct empleado emp[2];
      struct empleado *emp_ptr;

      emp[0].nombre="Angel";              emp[1].nombre="Pepita";
      emp[0].sexo='h';                    emp[1].sexo='m';
      emp[0].sueldo=20000;                emp[1].sueldo=20000;
      printf("  Empleado          Sexo          Sueldo\n");
      for(emp_ptr=emp;emp_ptr<=emp + 1;++emp_ptr)
         printf("  %s          %c          %d\n",
               emp_ptr->nombre,emp_ptr_>sexo,empptr->sueldo);
      }
```

Nueva versión del programa de nóminas.

(los enteros ocupan dos posiciones de memoria). Ahora puede aplicarse el operador "*" que da el contenido de la dirección a la que apunta lo que sigue a su derecha, obteniendo el mismo resultado en ambos casos. Lo descrito queda patente de un modo gráfico en la correspondiente figura.

Al igual que los otros tipos de variables, los arrays pueden ser inicializados también en el momento de su declaración. La única restricción sobre este punto es que tal inicialización sólo se podrá llevar a cabo cuando su tipo de almacenamiento sea externo o estático.

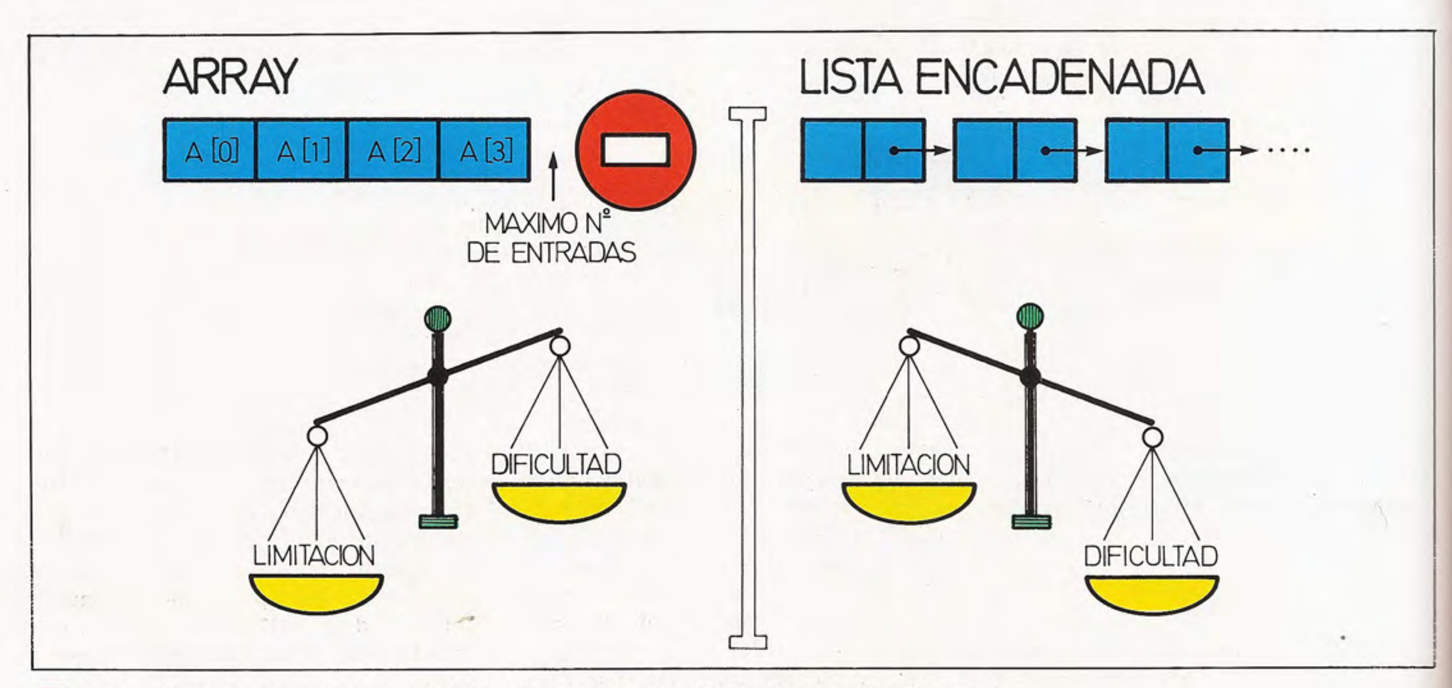
Una inicialización de array tiene el siguiente aspecto:

```
static int a[6]= {1,2,3,4,5,6} ;
```

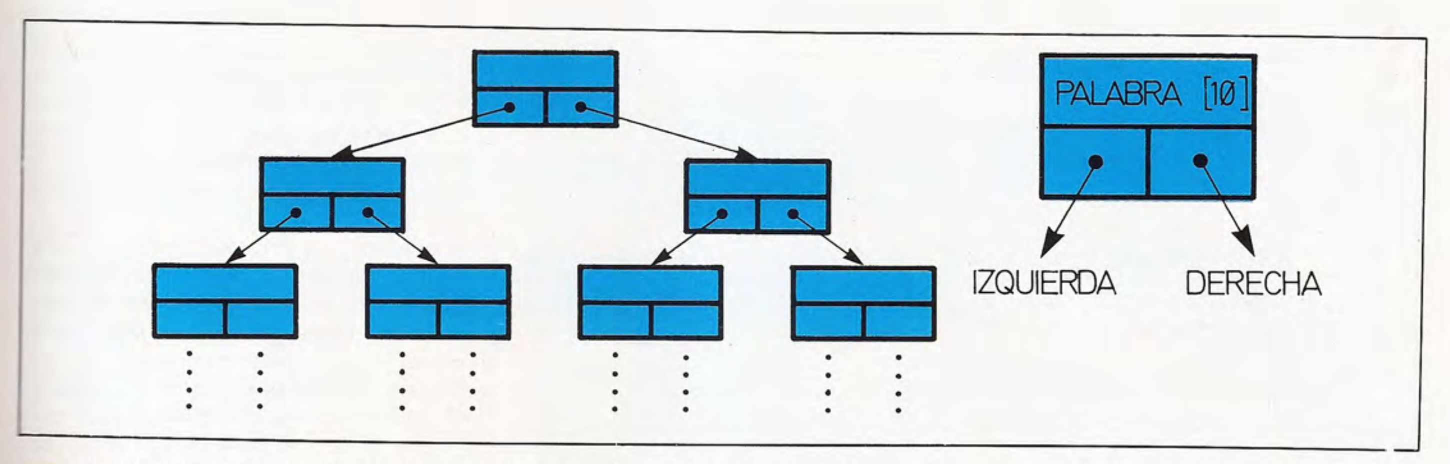
Como ejemplo del uso de arrays y de inicialización de uno de ellos con almacenamiento externo, en el cuadro adjunto se incluye un programa que calcula la suma de los elementos de un array.

Arrays de caracteres

Tienen su correspondencia en los llamados «string de caracteres» del BASIC; también en «C» se denominan así. Su



El problema derivado de operar con un número desconocido de datos de entrada puede solucionarse con una lista encadenada que sustituya al clásico array.



Un árbol binario tiene importantes aplicaciones en algunos problemas de ordenación de listas. También existen situaciones en las que una estructura de este tipo resulta ideal para representar los datos del problema.

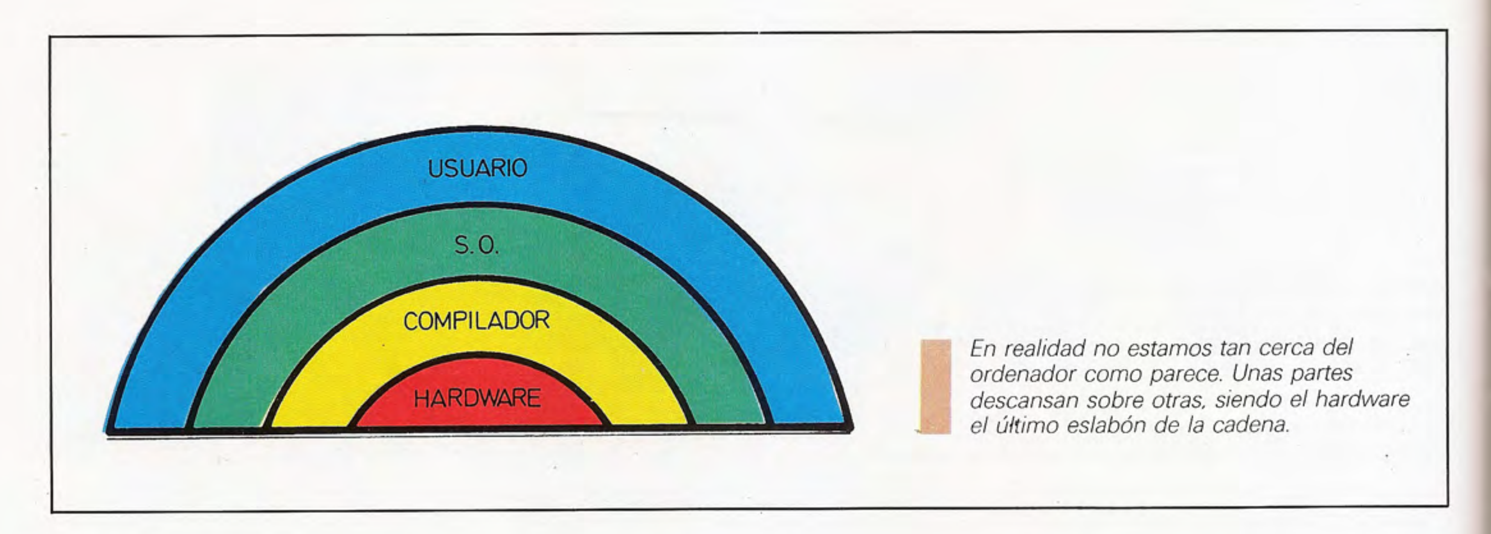
Para facilitar la actuación del C en cualquier entorno, las operaciones de entrada/salida son gobernadas por rutinas (como «printf») independientes del programa. Estas son manipulables por el usuario para adaptarlas a cada entorno de trabajo específico.

manipulación es análoga al resto de los arrays. Se representan encerrando los caracteres constituyentes entre dobles comillas:

"Esto es un string"

El ejemplo propuesto coincide con un string de 18 elementos. Sin embargo, entre las comillas sólo aparecen 17 letras. Ello se debe a que se introduce un carácter nulo (que se representa por '\0' y es un sólo carácter) al final del array, lo que permite manipularlo fácilmente, como se observa en el cuadro que acompaña al texto. El programa incluido en el referido cuadro escribe en columna las letras que componen el "string" y su equivalente ASCII.

Es preciso destacar la diferencia que existe entre 'x' y "x". La primera expresión representa a la letra x, mientras que la segunda es un string formado por dos caracteres: 'x' seguido por '\0'. Hay que recordar también que '\0' es completamente diferente a '0'. El primero se representa en binario como 00000000, mientras que la representación binaria del segundo es 00110000. Por esta razón, se puede decir que no existe propiamente el string nulo en C; el string "" consta en realidad de un carácter, el nulo



En realidad no estamos tan cerca del ordenador como parece. Unas partes descansan sobre otras, siendo el hardware el último eslabón de la cadena.

Utilidad y manejo de registros

A menudo nos vemos en la necesidad de agrupar datos de diversa naturaleza bajo un nombre común; en estos casos la estructura array se nos queda pequeña, ya que con ella sólo es posible agrupar datos de iguales características (ver figura).

Pensemos en un programa que necesite seguir la pista de distintos hechos ocurridos en distintas fechas. Para almacenar las fechas puede crearse un "struct" (registro o estructura) de la siguiente forma:

```
struct fecha {
   int dia;
   char mes [4];
   int año;
} ;
```

El equivalente para los conocedores del Pascal sería:

```
TYPE fecha=RECORD
                 dia: integer;
                 mes: array [1..3] of
                     char;
                 año: integer
             END
```

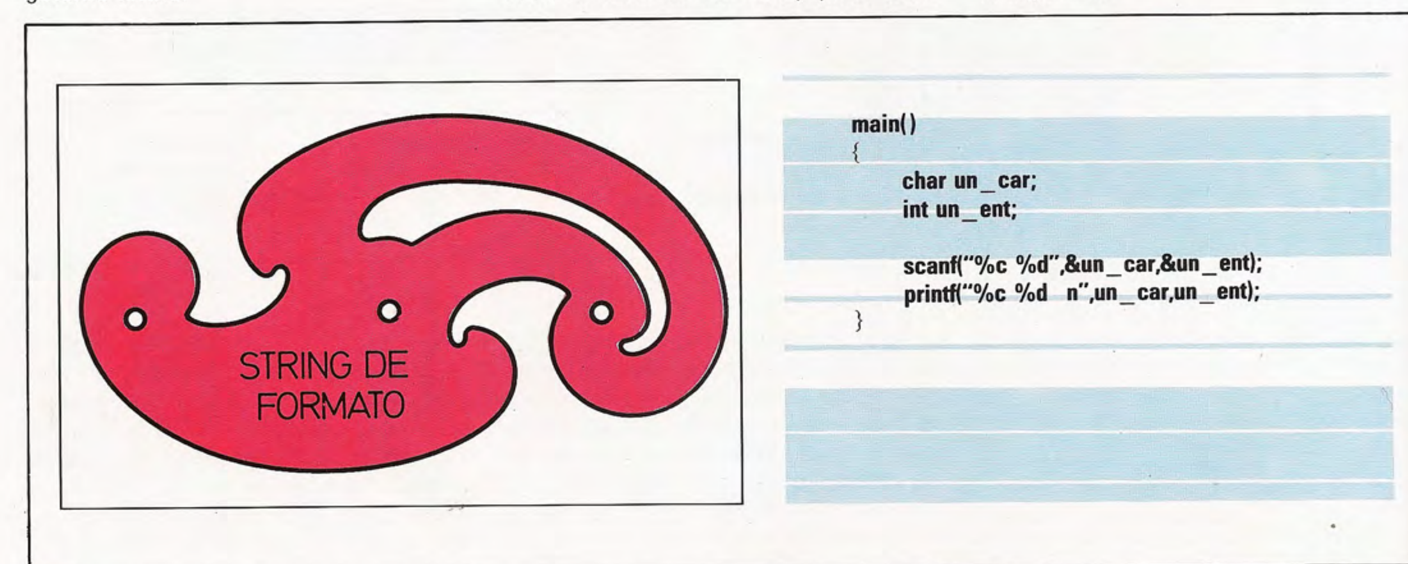
y cuidado con la dimensión de "mes": recuerde el carácter nulo que se introduce al final de un array para saber cuándo termina. En el lenguaje C es necesario contar con él en todo momento.

En el ejemplo se declara una estructura de nombre "fecha" y que contendrá tres elementos: dos de tipo entero y otro que es un array de caracteres.

Es importante reparar en que por el momento no se tiene aún ninguna variable de este tipo. Sólo se ha indicado al compilador que se prepare para lo que puede venir, al igual que ocurre en Pascal con la sección TYPE.

Las declaraciones de estructuras suelen ir al principio del programa, al igual que los "# define".

Para declarar una variable que sea una estructura del tipo "fecha" se hará lo siguiente



El string de formato es la plantilla según la cual se analizará la entrada realizada.

struct fecha nacimiento;

Ahora sí disponemos ya de la variable "nacimiento" que tiene la estructura comentada.

La forma de acceder a los elementos de esta variable es análoga a como se hace en Pascal:

```
nacimiento.día=21;
nacimiento.mes="mar";
nacimiento.año=1963;
```

Ahora se tiene en "nacimiento" la información sobre una fecha determinada: el 21 de marzo de 1963.

Al igual que con los arrays, se pueden inicializar "struct" directamente, siempre que estas sean externas o estáticas, nunca automáticas. Suponiendo que estamos en alguno de estos supuestos, la variable "nacimiento" podrá ser inicializada de la siguiente forma:

```
struct fecha nacimiento=
    {21,"mar",1963} ;
```

Las "struct" se pueden anidar entre ellas. Suponga el clásico ejemplo de la nómina de empleados de una empresa. Las primeras líneas de un programa en C para gestionar esta nómina podrían tomar un aspecto semejante al que aparece en la figura adjunta. Si se define una variable "emp", que designará a un empleado concreto, como sigue:

```
struct empleado emp;
```

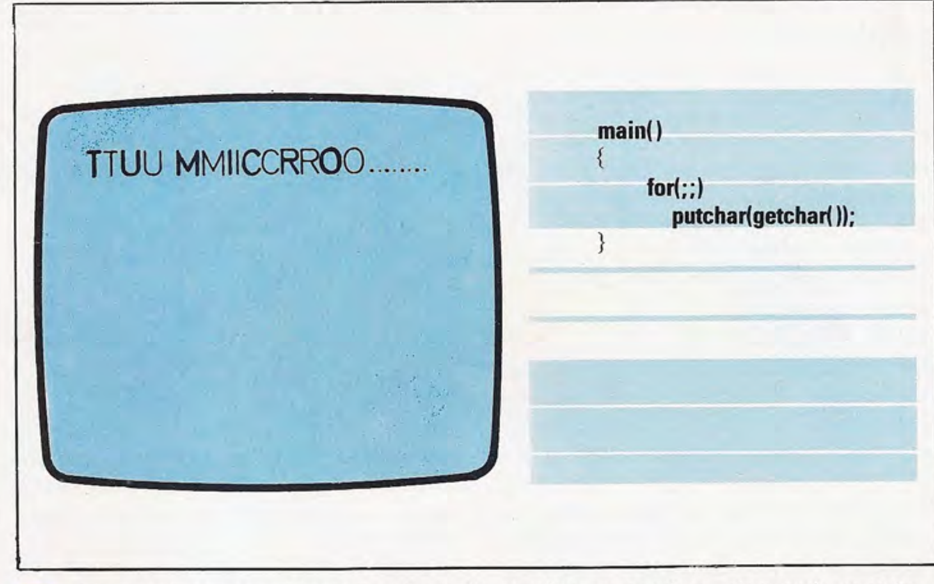
se accederá al mes de su nacimiento de la siguiente forma:

```
printf("%s n",emp.nacimiento.mes);
```

Por otra parte, las variables "struct" pueden ser declaradas en el momento de hacer la propia declaración de la "struct":

```
struct fecha
  int dia;          /*Se declaran «nacimiento» */
  char mes [4];     /*y «muerte» del tipo */
  int año;          /*«fecha»               */
      nacimiento,muerte;
```

Debido a la existencia de esta segunda forma de declaración, es muy importante no olvidar el ";" que sigue al último " } " de la declaración de la "struct".



Además de ilustrar el funcionamiento de «getchar» y «putchar», el ejemplo muestra la forma de realizar un bucle infinito en C. Los caracteres realzados corresponden a la entrada realizada por el teclado; los otros constituyen la salida del programa por pantalla.

Son numerosos los casos en los que un «matrimonio» entre arrays y "struct" resulta muy conveniente. En la figura adjunta se tiene un ejemplo de este tipo de unión.

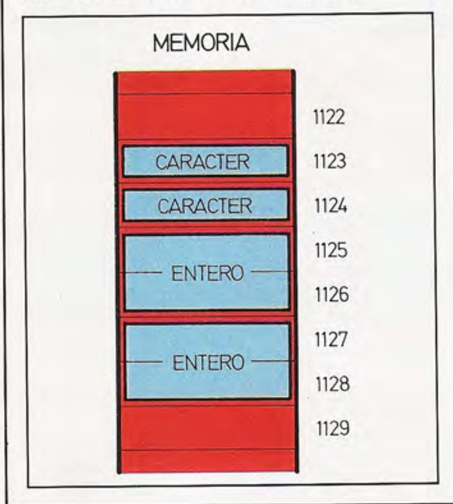
Hasta aquí todo habra resultado sencillo para los que ya conozcan el lenguaje Pascal. Sin embargo, pronto aparecerán nuevas emociones... Demos de nuevo la bienvenida a los punteros.

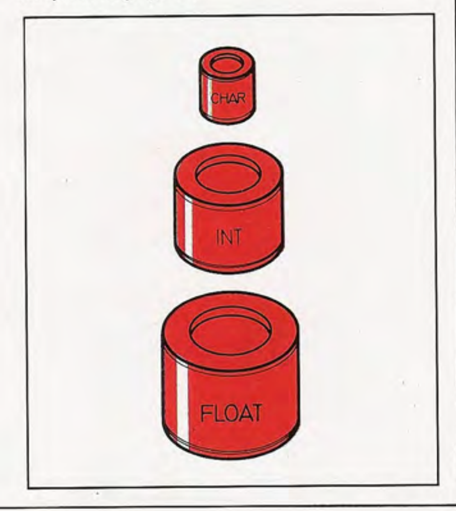
Los datos en memoria

En el lenguaje «C», los caracteres ocupan una posición de memoria, mientras que los valores enteros necesitan dos para su almacenamiento. El espacio ocupado por un valor en punto flotante, en simple o doble precisión, depende de las características del compilador.

En la figura aparece una zona de memoria que contiene dos caracteres en las posiciones 1123 y 1124 y dos enteros que comienzan en las posiciones 1125 y 1127, respectivamente.

Conocer el tamaño del espacio ocupado por las variables ayudará a entender mejor la filosofía de trabajo del lenguaje «C».





Cuando aparecen las restricciones

En las versiones actuales de los compiladores de C existen dos restricciones que están en vías de ser eliminadas en las próximas versiones del lenguaje. La primera restricción es que no se pueden manejar las "struct" como variables normales, en el sentido de que, teniendo dos estructuras "a" y "b" del mismo tipo, no es posible hacer "a=b", sino que habría que asignarlas campo a campo. Además —y aquí está la segunda restricción— tampoco se puede pasar una estructura completa como argumento de una función, aunque sí se pueden pasar campos individuales.

Por el momento, nuestra mejor arma está en los punteros: de igual forma que podíamos acceder a los elementos de un array considerando que el propio nombre del array era un puntero al elemento cero del mismo, podemos obtener la dirección de comienzo de nuestra estructura, y pasar esta dirección como parámetro a las funciones que lleven a cabo la tarea requerida.

Para afianzar todo esto nos remitimos al programa «Nueva revisión del programa de nóminas» que, básicamente, es una reforma del programa precedente.

Ambos entregan la misma salida, aunque el modo de operar es distinto. Observe la declaración:

```
struct empleado *emp__ptr;
```

Con ella declaramos a "emp__ptr" como un puntero hacia la estructura "empleado", de igual forma que en otras ocasiones hemos declarado punteros a enteros, caracteres o números reales (float).

Preste atención ahora al bucle "for". Este empieza inicializando el puntero "empt__ptr" a "emp"; esto es, se hace que "emp__ptr" señale al elemento cero del array "emp". Con este estado de cosas se entra en el cuerpo del bucle, que escribirá los campos de la estructura "emp[0]", los cuales han sido referenciados como sigue:

```
emp__ptr—>nombre
emp__ptr—>sexo
emp__ptr—>sueldo
```

en lugar de serlo a través del típico ".". Ello se debe a que ahora estamos utilizando un puntero para acceder a una estructura en vez de emplear su nombre, en cuyo caso utilizaríamos el ".". El símbolo "—>" está compuesto por los símbolos de resta y de «mayor que» y se trata como una unidad.

En otras palabras, como en la primera pasada, "emp__ptr" está apuntando a "emp[0]", las expresiones:

```
emp__ptr—>nombre    y
emp[0].nombre
```

son equivalentes.

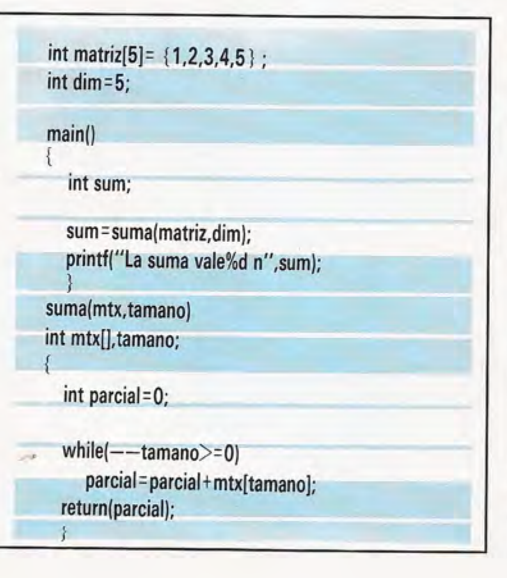
En la siguiente pasada, "emp__ptr" es incrementado en una unidad, es decir, lo suficiente como para que señale hacia "emp[1]" y el proceso se repite por última vez.



Aspecto de la función «scanf».

Suma de los elementos de un array

En el programa adjunto, se observa que el parámetro "mtx" de la función "suma" no contiene ningún subíndice en la declaración; basta con indicar que se trata de un array de enteros. Para el lenguaje C, esta información es suficiente. La condición del bucle "while" en "suma" está para abreviar, no para confundir. El efecto sería el mismo si tal condición fuera, simplemente, "tamaño>=0" y la primera sentencia del bucle fuera "——tamaño".



```
int matriz[5]= {1,2,3,4,5} ;
int dim=5;

main()
{
   int sum;

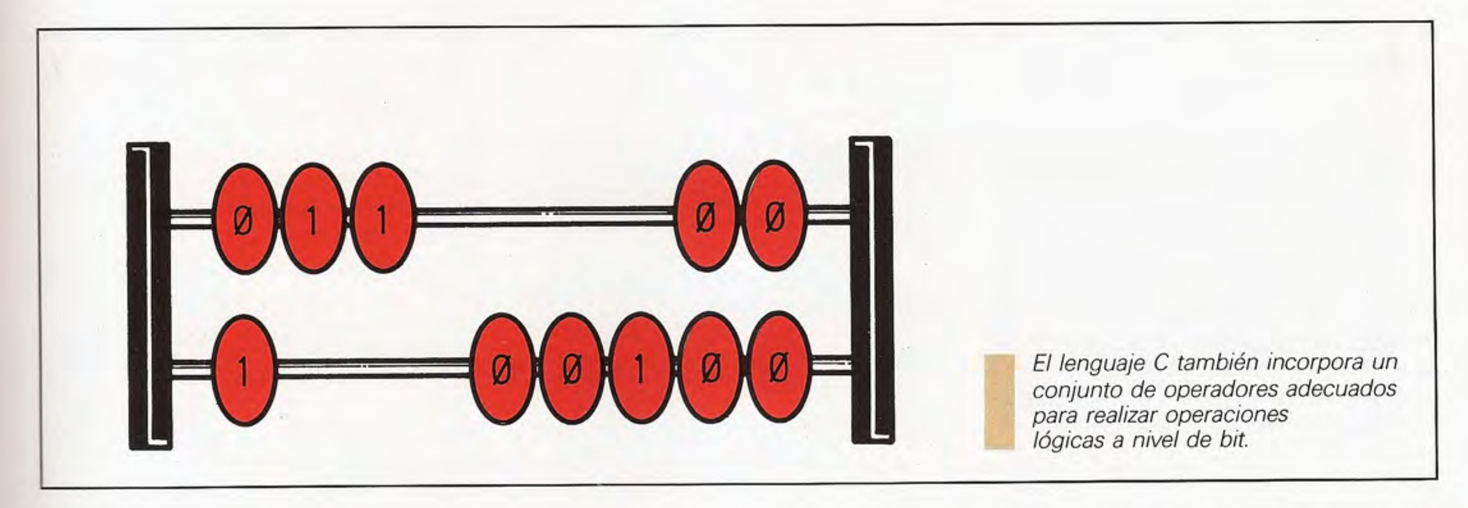
   sum=suma(matriz,dim);
   printf("La suma vale%d n",sum);
   }
suma(mtx,tamano)
int mtx[],tamano;
{
   int parcial=0;

   while(——tamano>=0)
      parcial=parcial+mtx[tamano];
   return(parcial);
   }
```

Estructuras autorreferenciadas

Este nombre tan poco expresivo se refiere a aquellas estructuras que sirven para crear y manipular las «listas encadenadas» y los «árboles binarios».

Estos dos «entes» informáticos (por darles un nombre) resuelven de manera elegante y sencilla problemas que de otra forma serían muy difíciles de tratar. Hay ocasiones en las que la cantidad de



El lenguaje C también incorpora un conjunto de operadores adecuados para realizar operaciones lógicas a nivel de bit.

datos de entrada para un determinado programa no se conoce de antemano. Si no disponemos de facilidades para crear listas encadenadas tendremos que recurrir a dimensionar un array, el cual en unas ocasiones se nos quedará pequeño y en otras nos sobrará espacio, desperdiciando memoria. Con una lista encadenada se ocupará sólo el espacio necesario en cada momento, pagando el precio de una mayor complejidad en el tratamiento de la información (ver figura).

La gestión de listas encadenadas en C es bastante más compleja que en Pascal, por lo que no la trataremos aquí. Simplemente mencionaremos que una lista encadenada se construye en torno a un "struct" como el que sigue:

```
struct elemento {
        int valor;
          struct elemento *siguiente;
} ;
```

en donde "elemento.valor" será el dato almacenado y "elemento siguiente" un puntero que servirá de enlace a la siguiente estructura.

Los árboles binarios se pueden construir en torno a una estructura como la siguiente:

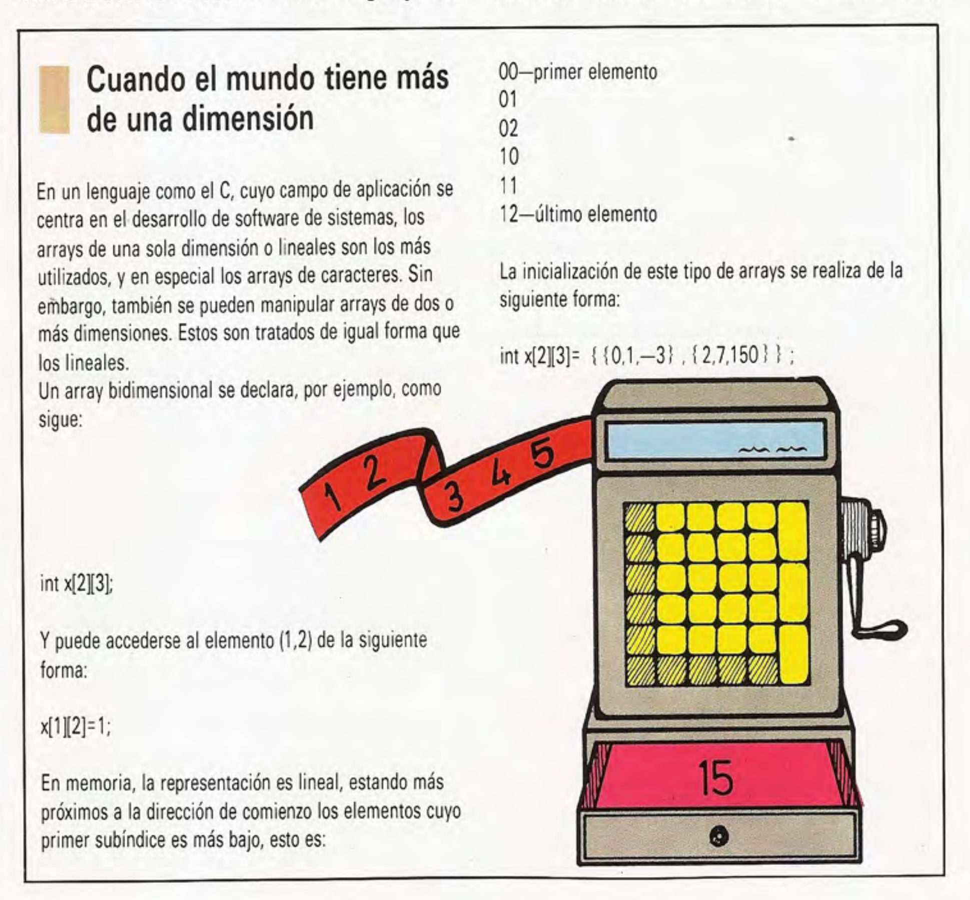
```
struct nodo {
    char palabra[10];
    struct nodo *derecha;
    struct nodo *izquierda;
} ;
```

Como se observa en la figura, un árbol binario está compuesto por nodos como el de arriba, enlazados por punteros a otros nodos.

La biblioteca de entrada/salida

Un buen número de las obras dedicadas a tratar un determinado lenguaje de programación comienzan explicando cómo realizar una salida por la pantalla a través de instrucciones del tipo PRINT, WRITELN o, en nuestro caso, "printf". Nosotros optamos por ese camino, pero pronto nos detuvimos dado que la entrada/salida en C es algo particular.

Las funciones que invocan la salida como las anterioremente citadas, o la entrada como son INPUT o READLN,



Cuando el mundo tiene más de una dimensión

En un lenguaje como el C, cuyo campo de aplicación se centra en el desarrollo de software de sistemas, los arrays de una sola dimensión o lineales son los más utilizados, y en especial los arrays de caracteres. Sin embargo, también se pueden manipular arrays de dos o más dimensiones. Estos son tratados de igual forma que los lineales.
Un array bidimensional se declara, por ejemplo, como sigue:

```
int x[2][3];
```

Y puede accederse al elemento (1,2) de la siguiente forma:

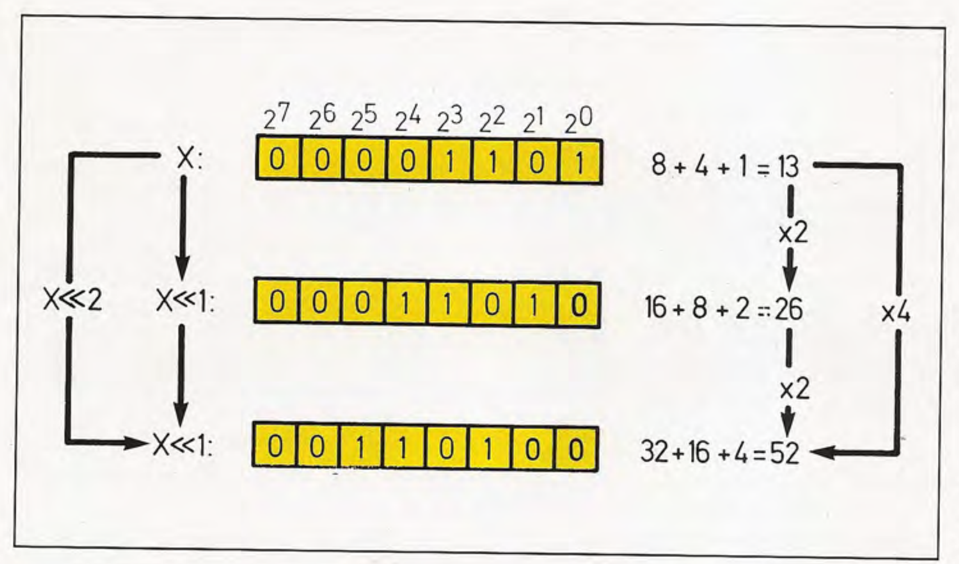
```
x[1][2]=1;
```

En memoria, la representación es lineal, estando más próximos a la dirección de comienzo los elementos cuyo primer subíndice es más bajo, esto es:

00—primer elemento
01
02
10
11
12—último elemento

La inicialización de este tipo de arrays se realiza de la siguiente forma:

```
int x[2][3]= { {0,1,—3} , {2,7,150} } ;
```



Cómo multiplicar de una forma rápida a base de ordenar el desplazamiento de los bits. Distintos desplazamientos dan distintos productos por potencias de dos. Por el extremo derecho se van introduciendo ceros a medida que se desplazan los bits del multiplicando.

constituyen una de las partes más complicadas de la implementación de un lenguaje, y están fuertemente condicionadas por el entorno del compilador: el sistema operativo.

Ya se ha comentado que el lenguaje C se diseñó para su uso en el entorno del sistema operativo UNIX, y, por lo tanto, la entrada/salida se efectúa apoyándose en las facilidades que proporciona éste. Previendo la posible utilización del C en otros entornos, las rutinas de entrada/salida —entre las que está "printf"— se encuentran en un fichero separado del resto del programa, el cual es gestionado por el «linker» para producir el código ejecutable. De esta forma, basta con manipular el fichero de las rutinas de entrada/salida adaptándolo a nuestro entorno concreto. Los nombres de estas rutinas no suelen cambiar, aunque sí su contenido.

La rutina que de forma general permite realizar las entradas por teclado se denomina "scanf". En la figura se reproduce el aspecto de esta función, el cual es muy parecido al de "printf", si bien el "string de formato" tiene un significado distinto. Ahora especifica el tipo de entrada que se va a teclear.

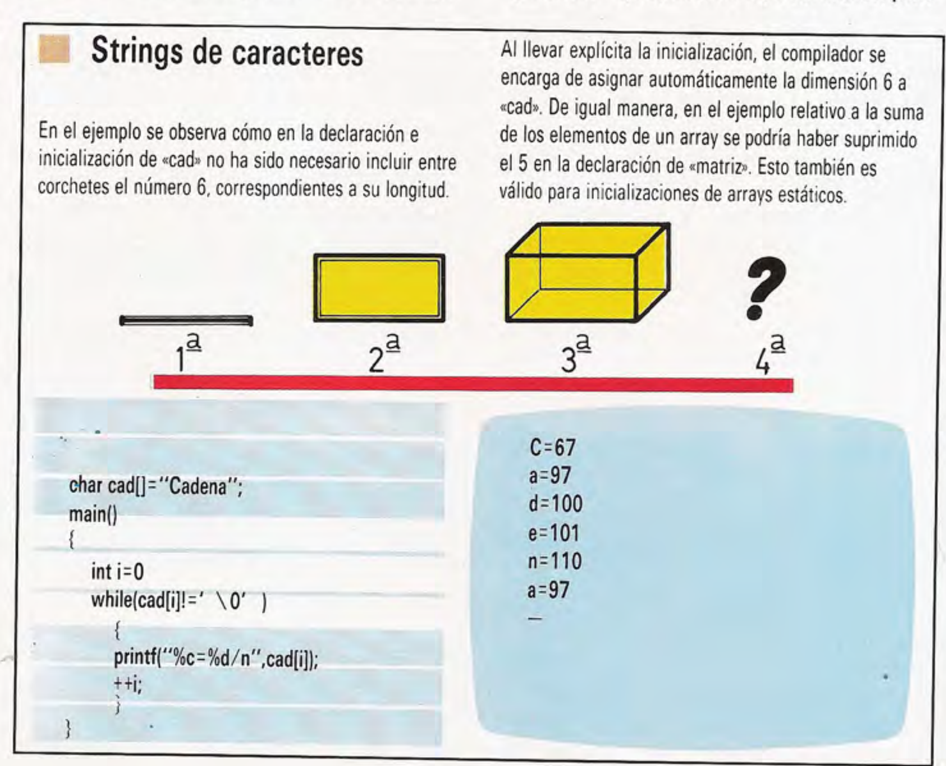
En el ejemplo ilustrado, el formato de "scanf" está indicando que la entrada estará compuesta por un carácter, un número indefinido de espacios en blanco (esto lo señala el espacio en blanco que hay entre "%c" y "%d" y un entero. La única forma de que una vez comenzada la ejecución de "scanf" nos salgamos de ella, es introducir los elementos especificados en el string de formato. Por ello, si se accionara la tecla de retorno de carro, o ENTER, a continuación del carácter "a", el programa seguiría ejecutando "scanf" esperando la segunda entrada.

Las variables hay que referenciarlas a través de su dirección, no de su contenido, por lo que es imprescindible el uso del operador "&" que da la dirección de la variable que lo sigue.

Normalmente, encontraremos a "scanf" demasiado torpe y difícil de manejar para nuestros propósitos concretos, por lo que lo mejor será redactar nuestras propias funciones de entrada/salida. En este sentido será útil la función "getchar", la cual detiene la ejecución de un programa hasta que se pulsa una tecla, devolviendo en ese momento el carácter correspondiente. Esta función no necesita ningún argumento, por lo que una llamada a la misma tiene el siguiente aspecto:

```
carácter=getchar();
```

El fiel acompañante de "getchar" es "putchar" que, cómo no, escribe en pan-



Strings de caracteres

En el ejemplo se observa cómo en la declaración e inicialización de «cad» no ha sido necesario incluir entre corchetes el número 6, correspondientes a su longitud.

Al llevar explícita la inicialización, el compilador se encarga de asignar automáticamente la dimensión 6 a «cad». De igual manera, en el ejemplo relativo a la suma de los elementos de un array se podría haber suprimido el 5 en la declaración de «matriz». Esto también es válido para inicializaciones de arrays estáticos.


```
char cad[]="Cadena";
main()
{
   int i=0
   while(cad[i]!=' \0' )
      {
      printf("%c=%d/n",cad[i]);
      ++i;
      }
}
```

```
C=67
a=97
d=100
e=101
n=110
a=97
_
```

talla el carácter especificado en su argumento. Por ejemplo:
putchar('a');

escribe el carácter 'a' en la pantalla. En la figura adjunta aparece un ejemplo de utilización de ambas funciones.

Todas estas funciones son tan sólo una parte de las que debe contener la biblioteca de entrada/salida del compilador que se esté manejando. Hay otras, si bien las principales son las mencionadas.

Las operaciones con bits

El lenguaje C incorpora una serie de operadores que permiten realizar operaciones lógicas con los bits de un entero o de un carácter, pero nunca con una variable de tipo "float" o "double". Estos operadores son los siguientes:

& and lógico
| or lógico
^ or exclusivo lógico (xor)
« desplazamiento a la izquierda
» desplazamiento a la derecha
~ complemento a uno

Así, por ejemplo:

```
c=n&OX7F;
```

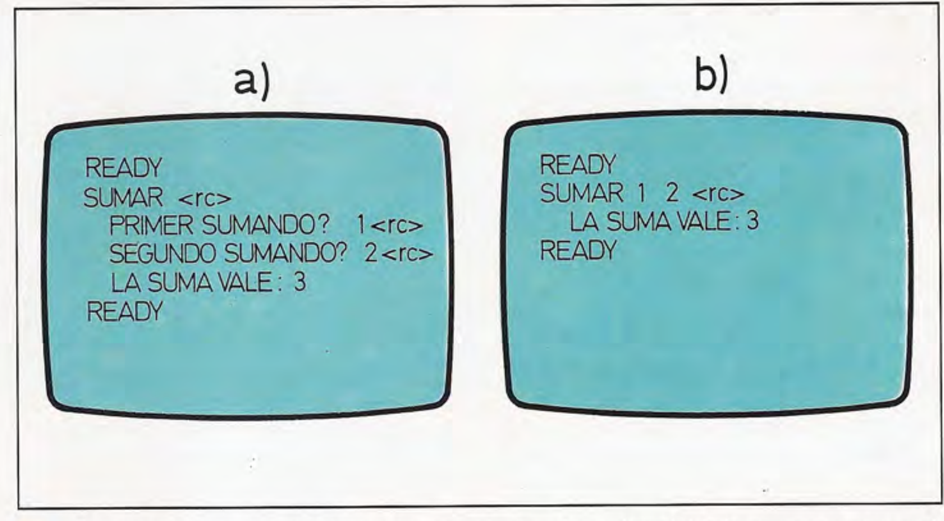
hace que se ponga a cero el bit más significativo de «n». Se observa que ha sido necesario anteponer al número hexadecimal 7F los caracteres "OX", precisamente para indicar al compilador que se trata de este tipo de constante.
Una expresión del tipo:

```
x«2;
```

hace que los bits de «x» sean desplazados dos posiciones a la izquierda, rellenando los espacios que van quedando a la derecha con ceros. El resultado es análogo al que se hubiera obtenido con la expresión:

```
x=x*4;
```

Pero la rapidez de ejecución del primer método es muy superior a la del segundo.



En ambos ejemplos de ejecución, «sumar» es el nombre del fichero que contiene el programa en C; programa que, previamente, habrá sido compilado.

El operador "~") se aplica de la siguiente forma:

```
~ x;
```

y, simplemente, cambia cada cero de «x» por un uno y viceversa.

Cómo decir mucho en pocas palabras

En varias ocasiones a lo largo de esta obra se ha mencionado que uno de los objetivos que se planteó el lenguaje C fue la concisión; es decir, expresar un cálculo en el menor número posible de líneas de listado. En la práctica resultará que para aquel que no conozca un poco el lenguaje, los programas en C se le antojarán escritos en lengua de otra galaxia.
Si todavía no está convencido de que el C es un lenguaje en el que la concisión es una característica primordial, observe la expresión siguiente:

```
x=1?y=2:y=3;
```

Para el que no haya visto este tipo de notación con anterioridad, le resultará muy difícil deducir que es completamente equivalente a la siguiente estructura:

```
if(x=1)
    y=2;
else
    y=3;
```

Hay que poner especial cuidado en no confundir los operadores "&&" y "&" ni "||" y "|". Un carácter de más o de menos tiene en C gran importancia.

La escasa importancia de ser «main»

Se ha comentado que «main» era como cualquier otra función del programa, y que sólo se diferenciaba en que siempre era la primera en ser ejecutada. Sin embargo, al contrario que la mayoría de las funciones, no lleva ningún parámetro... ¿Seguro? Vamos a ver que ello es falso: «main» sí puede llevar parámetros.

Para ejecutar un programa una vez compilado, basta teclear el nombre del fichero generado por el compilador. Pensando en un programa para sumar dos números, se puede tener la situación del primer ejemplo de la figura (caso a); en él, el programa acabará llegando a algún punto en el que pida los valores de los números a sumar para, posteriormente, continuar la ejecución. En el lenguaje C cabe la posibilidad de pasar datos al programa en el momento de llamarlo (segundo ejemplo de la misma figura, caso b), sin necesidad de esperar a que ésta los pida.

En el segundo caso, «main» tiene dos argumentos. Su cabecera tendría el siguiente aspecto:

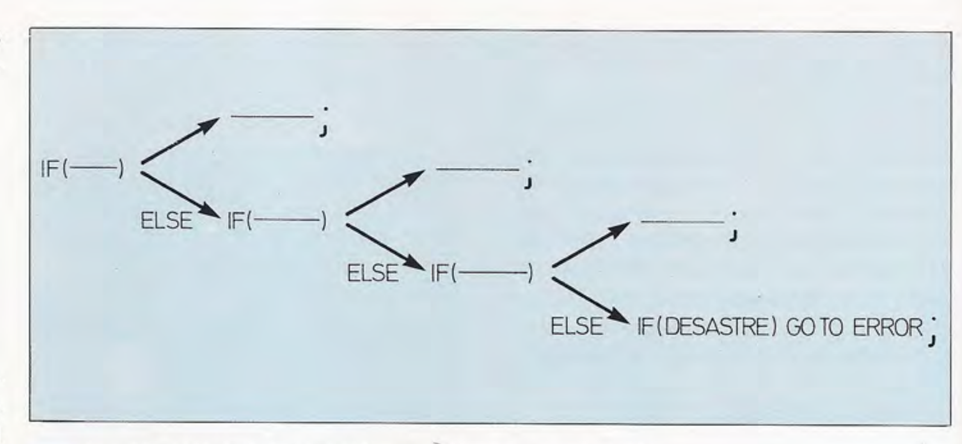
```
main(argc,argv)
int argc;
char*argv[];
{
        .........
        .........
}
```

"argc" es el número de palabras (grupos de caracteres separados por blancos) que había en la línea que invocó al programa; en nuestro caso, como se ha tecleado:

sumar 1 2<r.c.>

su valor sería tres."argv" es un puntero a un array de strings de caracteres el cual contiene tales palabras. En nuestro ejemplo su contenido es:

```
argv[0] .... "sumar"
argv[1] .... "1"
argv[2] .... "2"
```



La mejor victoria es una retirada a tiempo... aunque sea esgrimiendo la orden GOTO.

Puede utilizarse esta información para calcular los valores a sumar sin necesidad de tener que incluir en el programa las rutinas necesarias para realizar la entrada por teclado.

Realmente, no tiene tanta importancia llamarse «main».

La unión hace la fuerza

En C existe un tipo de variable llamado «union» con el cual se pueden definir variables que, en distintos momentos, pueden contener por ejemplo, un entero, un carácter o un real, aun teniendo el mismo nombre. La «union» son muy parecidas a las «struct» en cuanto a sintaxis, manejo y restricciones. Por ejemplo:

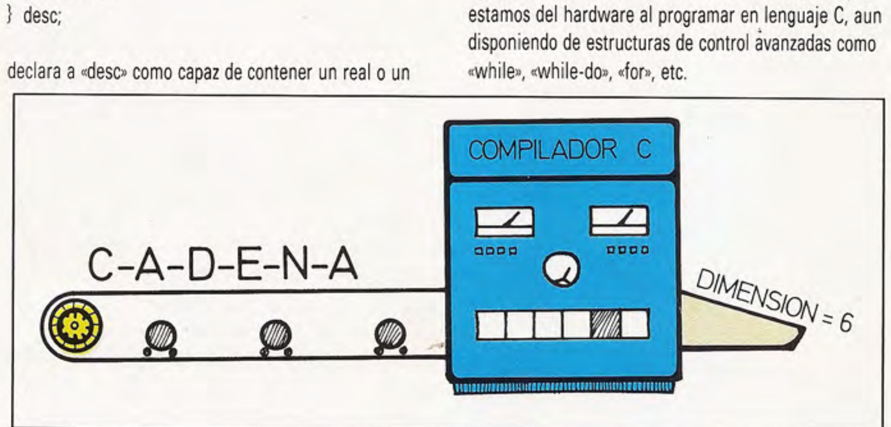
```
union ejemplo {
     int entero;
     float real;
} desc;
```

declara a «desc» como capaz de contener un real o un entero; ello equivale a tener en un mismo programa las declaraciones:

```
int desc;
float desc;
```

Cualquiera de estos dos tipos (que sabemos que ocupan distinta cantidad de memoria) podrá ser asignado y utilizado en expresiones con «desc».

Dedicar unas líneas tan sólo a la «union» en un curso introductorio está fuera de lugar, pero el objetivo de hacerlo no es otro que resaltar de nuevo lo cerca que estamos del hardware al programar en lenguaje C, aun disponiendo de estructuras de control avanzadas como «while», «while-do», «for», etc.



La sentencia "goto"

Los saltos incondicionales que representan las sentencias "goto" son innecesarias en un lenguaje estructurado como el C, aunque hay ocasiones en las que llevar al límite este «estructuralismo» sólo puede acarrear dolores de cabeza; una sentencia "goto" en el momento y lugar adecuados puede ahorrar mucho trabajo.

Un ejemplo interesante sobre el uso de esta sentencia lo constituye la necesidad de «salir» de una estructura muy anidada, como la representada en la figura adjunta. Si al llegar a un determinado nivel nos encontramos con un error que obligue a abandonar todo el proceso, lo mejor es atajar por lo fácil con un "goto".

La sentencia "goto" tiene el siguiente aspecto:

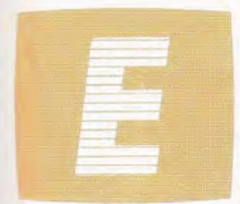
goto<etiqueta>;

donde <etiqueta> es un nombre como los utilizados para las variables, seguido del signo":". Al ejecutarse el "goto", el control bifurcará a las sentencias que siguen a la etiqueta.

Cabe recordar que el Pascal, el lenguaje estructurado por excelencia, incluye la sentencia "GOTO", lo cual es una buena excusa para utilizar esta orden en otros lenguajes no tan estructurados.

COBOL

Un pionero de los lenguajes informáticos



l COBOL es un lenguaje que, aunque tenga una escasa presencia en el terreno de los ordenadores personales, ocupa un puesto muy relevante en el panorama pasado y presente de la informática.

El propósito del COBOL

El COBOL lleva la orientación que se le quiso dar desde su nacimiento en su propio nombre. En efecto, la palabra COBOL está formada por las iniciales de «COmmon Business Oriented Language», esto es, «lenguaje general orientado a los negocios».

Una aplicación típica del COBOL puede consistir en la gestión de la nómina de empleados de una empresa. La tarea encomendada al programa será que, cada vez que sea ejecutado a fin de mes, calcule el sueldo que va a recibir cada empleado según sus aumentos, cotización, etc., imprima el cheque que recibirá cada trabajador y dos resguardos, uno para la emprea y otro para el interesado. Esta aplicación se caracteriza por:

1. Bajo volumen de datos de entrada. De hecho, el programa sólo necesitará nuevos datos si se ha contratado a alguien por primera vez en el transcurso del mes; aunque si la empresa es grande, el número de nuevos contratados puede resultar elevado.
2. Poco volumen de cálculo. Las operaciones que hay que realizar con los sueldos se reducen a sumas, restas y algún tanto por ciento.
3. Gran volumen de salida. Aunque en la impresión de los cheques existen pocas variantes, no sucede lo mismo con los resguardos, en los que pueden ir especificados los trabajos realizados, horas ocupadas y un sinfín de datos aplicables tan sólo a una parte de los empleados.

El lenguaje COBOL nació en 1959, orientado específicamente a la gestión administrativa y a los negocios. El objetivo de sus creadores era convertirlo en un lenguaje universal para todos los ordenadores.



En lenguaje COBOL (COmmon Bussines Oriented Language) pueden encontrarse miles de aplicaciones orientadas a la gestión administrativa. El soporte de almacenamiento (disquete, casete, etc.) varía según el ordenador a que el programa esté destinado.

Estas tres características son en gran parte comunes a la mayoría de las aplicaciones informáticas que requieren los «negocios». COBOL se ajusta a este tipo de requerimientos de una forma que no lo hacen otros lenguajes; esta es una de la razones por las que su importancia se mantiene hoy en día.

A vueltas con la historia

Aunque nuestro protagonista es apenas cuatro años más joven que el FORTRAN, encontramos en él características que lo alejan mucho de su compañero científico. Su nacimiento se produjo a raíz de una reunión convocada por el Departamento de Defensa de los Estados Unidos y en la que estaban representadas las mayores empresas informáticas de la época (algunas de las cuales sobreviven hoy, como IBM, Honeywell o Burroughs), instituciones y el ejército. Esta convocatoria recibió el nombre de CODASYL, siglas de COnference on Data SYstems Languajes (Conferencia sobre los lenguajes para sistemas de datos).

De CODASYL surgió el COBOL 60. Posteriormente fueron apareciendo variantes de este COBOL, manteniendo siempre la misma estructura y sentencias aunque añadiendo otras nuevas; así se llegó hasta el COBOL ANSI 74 y el COBOL 80. Más recientemente, han aparecido el CISCOBOL, con facilidades para el manejo de pantallas, y el RM-COBOL, especialmente diseñado para su uso en sistemas basados en microprocesador.

Con el COBOL nació un concepto que se ha mantenido en la mayoría de los lenguajes que aparecieron con posterioridad: la separación entre el programa propiamente dicho y los datos del mismo. Esto trae consigo indudables beneficios en la depuración y mantenimiento de programas (ver cuadro adjunto).

En la actualidad, el COBOL mantiene su vigencia debido en parte al gran número de aplicaciones que han sido desarrolladas con él y a la experiencia adquirida por los programadores; aunque también constituye una razón primordial el hecho de que los requerimientos de las aplicaciones de negocios se adaptan muy bien a sus prestaciones.

Por lo demás, la circunstancia de que una empresa cambie el ordenador que está utilizando no significa tirar por la ventana todo el software desarrollado hasta la fecha. Los programas escritos en COBOL tienen el «don» de la transportabilidad, esto es, son fácilmente implementables sobre distintas máquinas.

Elementos del lenguaje

Como todos los lenguajes de programación, el COBOL consta de una serie de elementos (variables, constantes, verbos), utilizables para la escritura y

definición del programa, y un conjunto de reglas sintácticas que rigen la combinación de los anteriores para la formación de sentencias. Puesto que un compilador está diseñado para reconocer única y exclusivamente un grupo determinado de comandos ordenados de una forma precisa, cualquier cambio, voluntario o inadvertido, en alguno de estos elementos ocasionará la generación de un mensaje de error. Por supuesto, la corrección sintáctica del programa indica únicamente que no se ha violado ninguna de las reglas de escritura, pero no que vaya a funcionar correctamente.

Dentro de la estructura del COBOL pueden distinguirse los siguientes elementos fundamentales:

* Palabras reservadas (ver tabla), que el programador no puede nunca alterar o modificar, y que reciben el nombre de «verbos». Su misión es indicar al ordenador la función que debe realizar con los datos que contenga la instrucción en la que se encuentran. En general, se trata de palabras o abreviaturas anglosajonas, como ADD (sumar), WRITE (escribir) o PERFORM (realizar).

* Nombres proporcionados por el usuario para la identificación de variables, constantes o zonas del programa. Deben, a ser posible, indicar la naturaleza de su misión (es, por ejemplo, mucho más clarificadora una variable llamada SALDO que otra que se denomine S), y cumplir, entre otras, las siguientes reglas: emplear únicamente caracteres alfanuméricos (A-Z y 0-9) o guiones, en número menor o igual que 32, y no emplear guiones al principio o al final, ni contener espacios en blanco.

* Símbolos aritméticos, condicionales y de puntuación, empleados para distintas tareas según su condición. El primer grupo comprende los operadores aritméticos fundamentales (+, −, *, /, y ** (exponenciación), el segundo los operadores de relación (<, > e =), y el tercero los distintos elementos empleados en la definición de sentencias del lenguaje: el punto, utilizado al final de todas las instrucciones; la coma, separadora de operandos dentro de una instrucción; el punto y coma, empleado en la separación de cláusulas; las comillas, que marcan los literales no numéricos, y los paréntesis, para enmarcar subíndices o expresiones.

* Literales, que puede ser de dos tipos: numéricos y no numéricos. Cada uno de ellos ha de cumplir una serie de normas, como son el no emplear más de dieciocho dígitos los primeros y más de ciento veinte los segundos, e ir, estos últimos, encerrados entre comillas.

Un caso particular de literales no numéricos lo constituyen las llamadas constantes figurativas, ZERO, ZEROES, SPACE, SPACES y QUOTE, que no son sino constantes dotadas de nombre propio.

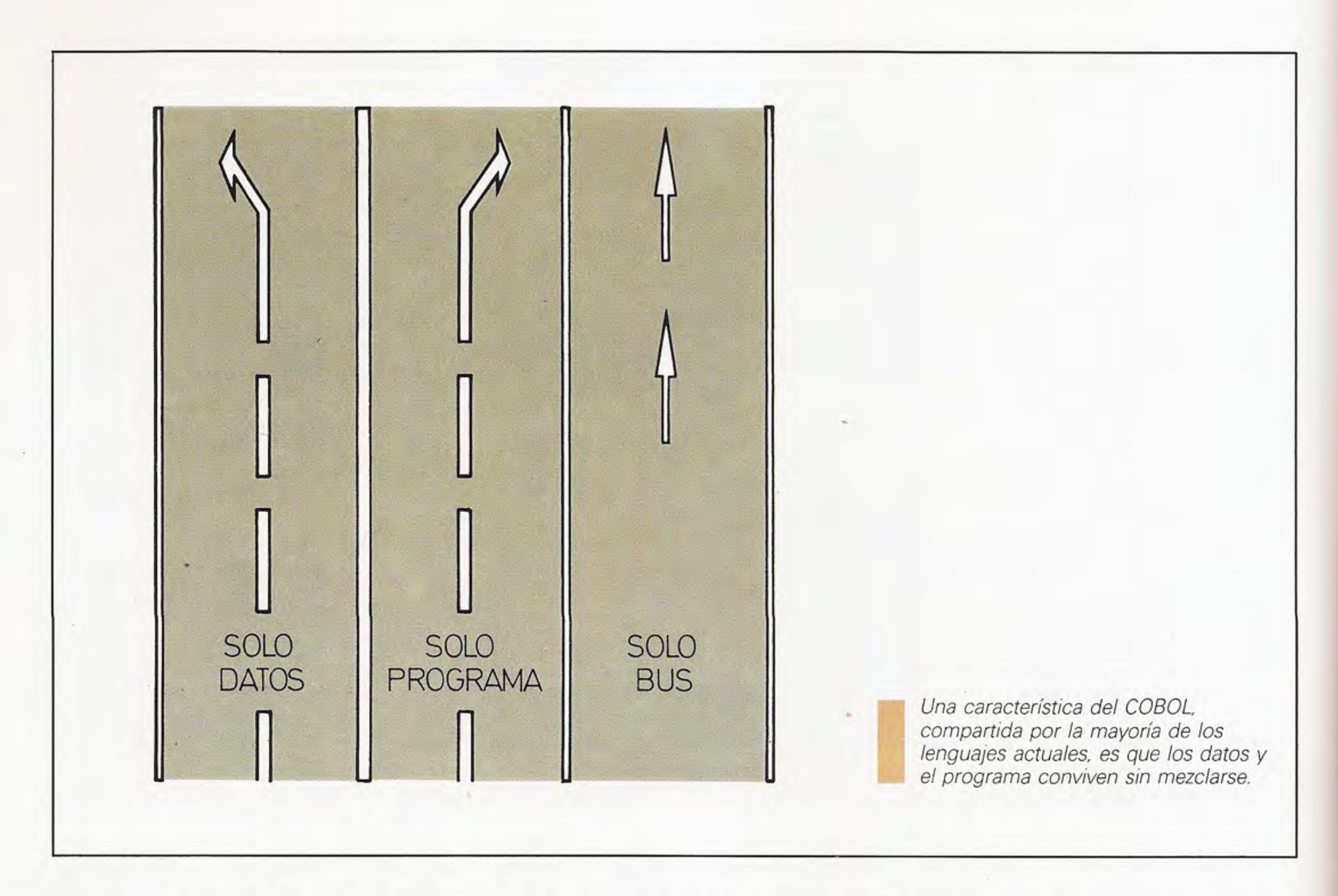
Estructura de un programa COBOL

Un programa en lenguaje COBOL se encuentra jerarquizado de la siguiente forma: Divisón, Section, Paragraph, Sentence, Statement, Word y Character.

En la figura adjunta se muestran las cuatros «divisions» (zonas o divisiones) en las que se divide todo programa COBOL, intentando buscar un equivalente Pascal a las mismas. Estas cuatro divisiones deben aparecer siempre, y en el mismo orden que refleja la figura. Sus



En la actualidad existen numerosas versiones o dialectos del COBOL 60 original. Cada uno de ellos supone una mejora o adaptación a necesidades del usuario, pero siempre dentro de la orientación primera: la gestión administrativa y los negocios.



Una característica del COBOL, compartida por la mayoría de los lenguajes actuales, es que los datos y el programa conviven sin mezclarse.

propósitos son los que se describen a continuación:

* *Identification Division*

El compilador no producirá ningún código objeto a partir de ella. Es algo parecido a tener una serie de líneas REM en un programa BASIC. Su contenido especifica datos concernientes al programador, propósito y estructura del programa, fecha de realización, etc. Es importante que esta zona esté claramente especificada y sea lo más detallada posible, de forma que pueda ahorrarse tiempo a la hora de modificar el programa, bien sea el mismo u otro equipo de programadores.

El formato de esta división es:

IDENTIFICATION DIVISION

PROGRAM-ID. nombre-del-programa.

AUTHOR. comentarios-sobre-el-autor.

INSTALLATION. comentarios-sobre-instalación.

DATE-WRITTEN. fecha-de-creación.

DATE COMPILED. fecha-de-compilación.

SECURITY. comentarios-sobre-seguridad.

REMARKS. comentarios-en-general.

El nombre del programa es un conjunto de caracteres que servirá para la identificación del mismo. Su formato es similar al de los nombres de procedimientos y etiquetas. Como caso general, sólo los ocho primeros caracteres son tenidos en cuenta por el compilador, que simplemente ignora los demás, caso de que los haya.

De todas estas líneas, sólo las dos primeras son obligatorias. El resto puede ser incluído o no.

* *Environment Division*

Esta zona del programa es la única que depende de la máquina concreta sobre la que se vaya a compilar y ejecutar el programa. En ella se describen los ordenadores que van a tratar tanto el código fuente como el programa objeto, y se definen las distintas unidades periféricas que contendrán los ficheros con los que ha de trabajar el programa, así como los periféricos de salida (impresoras, pantallas, etc.). Estas dos funciones se corresponden con las dos secciones en que se divide la Environment Division: la Configuration Section, en la que se identifican los programas fuente y objeto, y la Input-Output Section, que relaciona todos los dispositivos empleados por el programa.

El formato de esta división es:

ENVIRONMENT DIVISION

CONFIGURATION SECTION

SOURCE-COMPUTER. ordenador-fuente.

OBJECT-COMPUTER. ordenador-objeto.

SPECIAL-NAMES. nombres-especiales.

INPUT-OUTPUT SECTION.

FILE-CONTROL. control-de-ficheros.

I-O-CONTROL. control-de-entrada-salida.

En la CONFIGURATION SECTION deben aparecer, obligatoriamente, las dos primeras líneas, que indican, respectivamente, el ordenador en que se va a compilar el programa, y aquél en el que se va a ejecutar. La línea SPECIAL-NAMES reseña los nombres simbólicos utilizados por el programador, definitivamente dependientes de la marca y modelo del ordenador.

La INPUT-OUTPUT SECTION es opcional, aunque si aparece deberá ir seguida obligatoriamente por la zona FILE-CONTROL, que consta de tantas sentencias SELECT ASSIGN (asignación de un nombre de fichero a un dispositivo externo) como periféricos se usen. La línea I-O-CONTROL especifica procedimientos y/o técnicas especiales de tratamiento de información. Puede omitirse en caso de que no sea necesaria.

* *Data Division*

Describe los ficheros, las estructuras de datos y las variables que van a ser procesadas: longitud de los campos en variables de tipo carácter (ALPHABETIC), posición del punto decimal en variables de tipo numérico (NUMERIC), organización de los registros, etc.

La Data Division proporciona las características y organización de los datos de una manera que es, en gran medida, independiente del equipo que se utilice (cuya definición está en la Environment Division) y casi por completo independiente de la forma en la que se van a procesar los datos (especificada en la Procedure Division).

Esta zona del programa se divide a su vez en cuatro secciones: la File Section, en la que se definen según ciertas reglas todos los ficheros que intervienen en el programa, la Working-Storage Section, en la que se declaran los elementos que necesita el programador para su trabajo (buffers, contadores, acumuladores, tablas, etc.), la Linkage Section, para la especificación de las variables de enlace con el programa principal (se emplea únicamente en los subprogramas), y la Report Section, donde se describe la composición y el formato de los datos de salida. La estructura de esta división es:

```
DATA DIVISION
FILE SECTION
  descripción-de-fichero
    descripción-de-registro
WORKING-STORAGE SECTION
descripción-de-identificadores
descripción-de-grupos
```

Cualquier de las dos secciones puede aparecer o no.

Dentro de la FILE-SECTION debe haber tantas descripciones de ficheros como ficheros hayan sido mencionados en la ENVIRONMENT DIVISION; en cada una de ellas pueden describirse tantos registros como se necesite.

Cada entrada de descripción de un fichero se identifica con las iniciales FD (File Description). En ella debe aparecer obligatoriamente la cláusula LABEL RECORDS (etiquetas que posee el archivo, bien STANDARD, bien OMMITED, o generadas por el usuario), a la que pueden o no seguir algunas de las siguientes:

RECORDING MODE, para indicar el modo de grabación física del fichero.

BLOCK CONTAINS xxxx RECORDS, que indica el número de registros lógicos contenidos en uno físico.

RECORD CONTAINS xxxx CHARACTERS, donde se define el número de caracteres que debe poseer cada registro, y

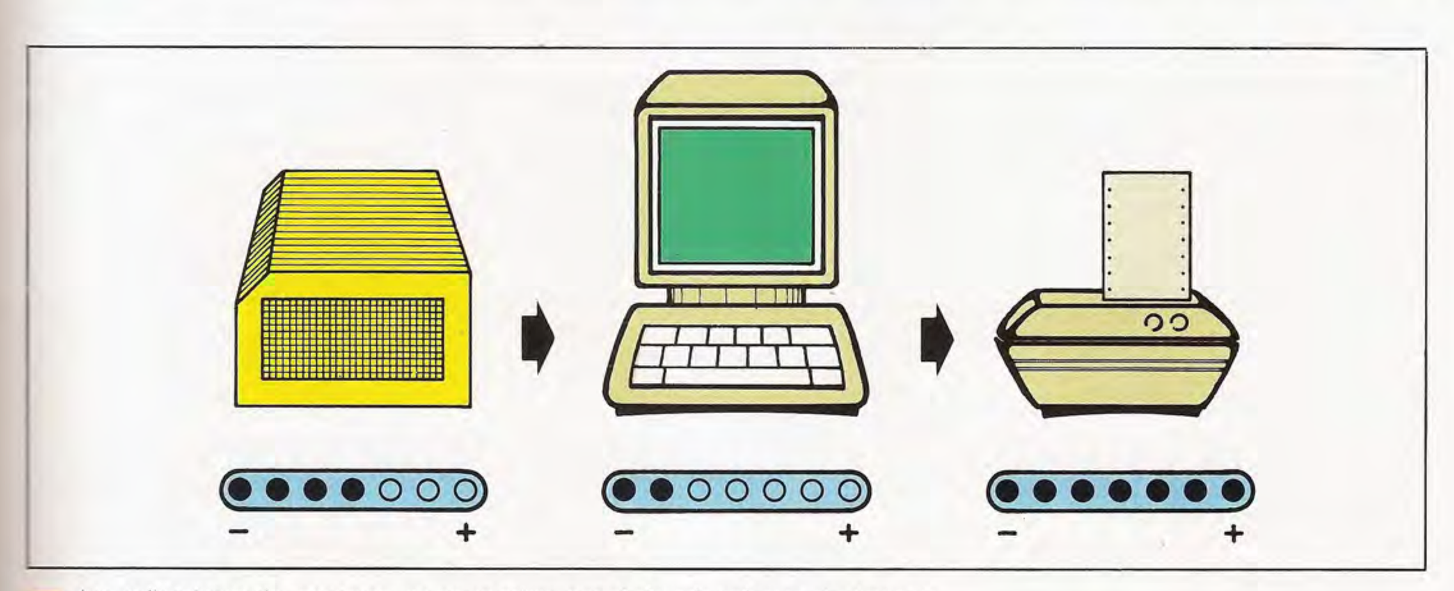
DATA RECORD(S) IS(ARE) nombre-de-registro, que determina el nombre de los registros asociados al fichero que se está declarando.

Es dentro de la WORKING-STORAGE SECTION donde se describen los distintos campos e identificadores utilizados por el programador. Para ello se emplean algunas de las siguientes cláusulas:

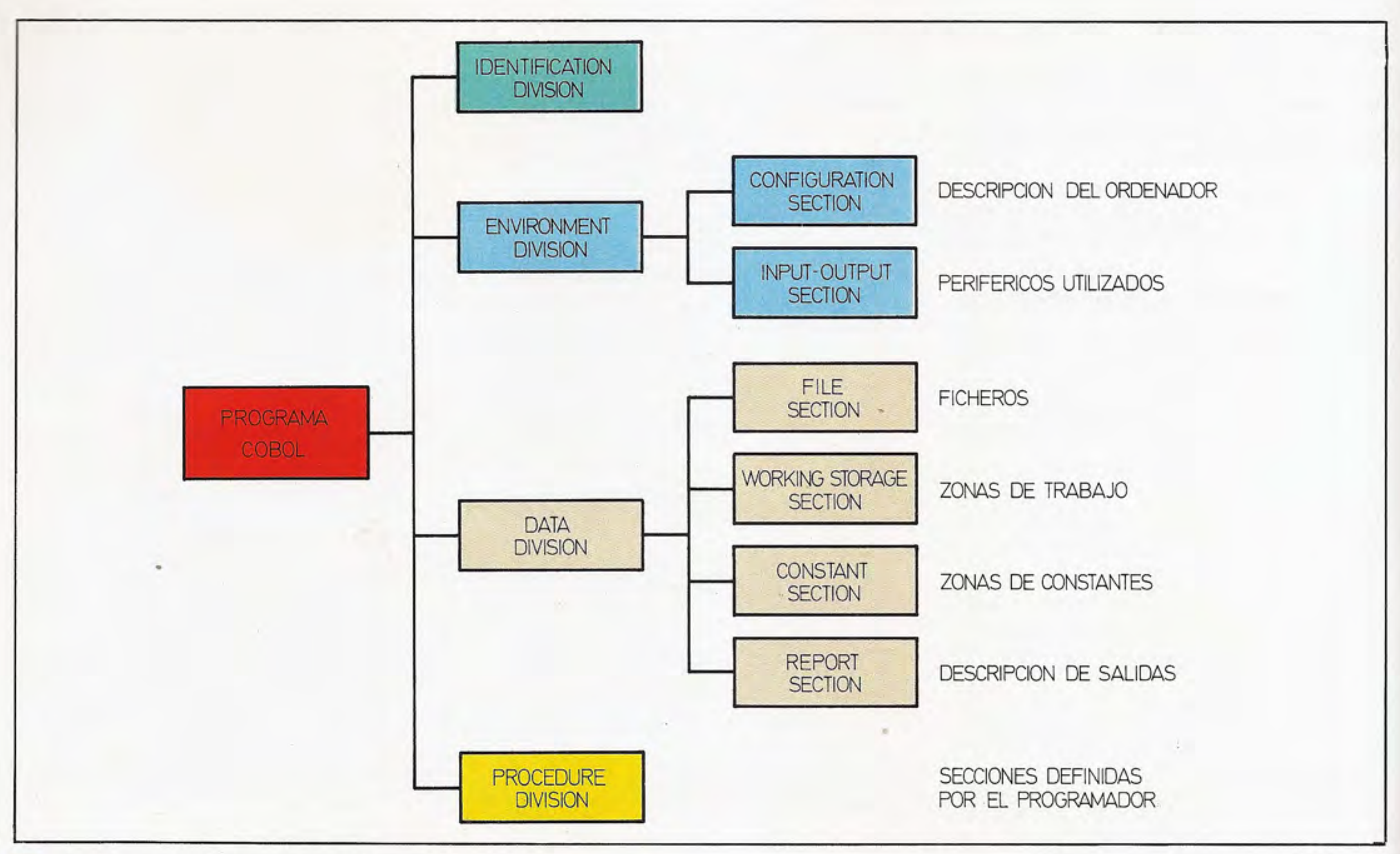
PICTURE: tipo (numérico, alfabético o alfanumérico) y longitud del identificador.

VALUE: inicialización de la variable que preceda a esta palabra clave.

JUSTIFIED: ajusta el contenido de una



Las aplicaciones de «negocios» se caracterizan por el elevado volumen de datos de salida comparado con el de entrada o el de cálculo.



Estructura genérica de un programa confeccionado en lenguaje COBOL.

variable a la derecha o izquierda del campo que tenga asignado.

USAGE: formato de almacenamiento de la variable en memoria. Depende de la máquina con la que se esté trabajando.

BLANK WHEN ZERO: se aplica cuando se desee que un campo con valor cero se escriba como blancos.

OCCURS: indica el número de veces que se repite un identificador dentro de una tabla.

REDEFINES: permite cambiar el nombre a una variable o estructura definida con anterioridad.

SYNCHRONIZED: asegura el alineamiento correcto de los datos en un fichero.

Las otras dos secciones (LINKAGE y REPORT) pueden aparecer o no en la definición de esta división.

* *Procedure Division*

Esta zona del programa es la que especifica las distintas acciones que hay que realizar sobre la Data Division. En lugar de secciones, esta parte se encuentra dividia en párrafos, cada uno de los cuales debe ser identificado por el programador mediante una etiqueta o nombre. Es dentro de cada uno de estos párrafos donde se indican las diferentes transformaciones que deben aplicarse a los datos del programa.

En la figura está representada la estructura final que, a grandes rasgos, presenta un programa COBOL.

Tipos de sentencias

Una instrucción es una combinación sintácticamente válida de palabras y símbolos, que comienza por un verbo COBOL. Los «verbos» son palabras reservadas del lenguaje (ver tabla), por lo que el programador no puede modificarlos bajo ningún concepto. Como es habitual en todos los lenguajes, estas palabras no pueden emplearse para dar nombre a constantes o variables, o el compilador será incapaz de ejecutar su tarea.

COBOL utiliza los siguientes tipos de instrucciones:

* *Instrucciones de entrada-salida*

Dentro de este grupo se encuentran todas las sentencias encargadas de la manipulación de ficheros: OPEN (para abrir un fichero), CLOSE (indica que ese fichero no va a ser usado de nuevo), READ, WRITE y REWRITE (para la lec-

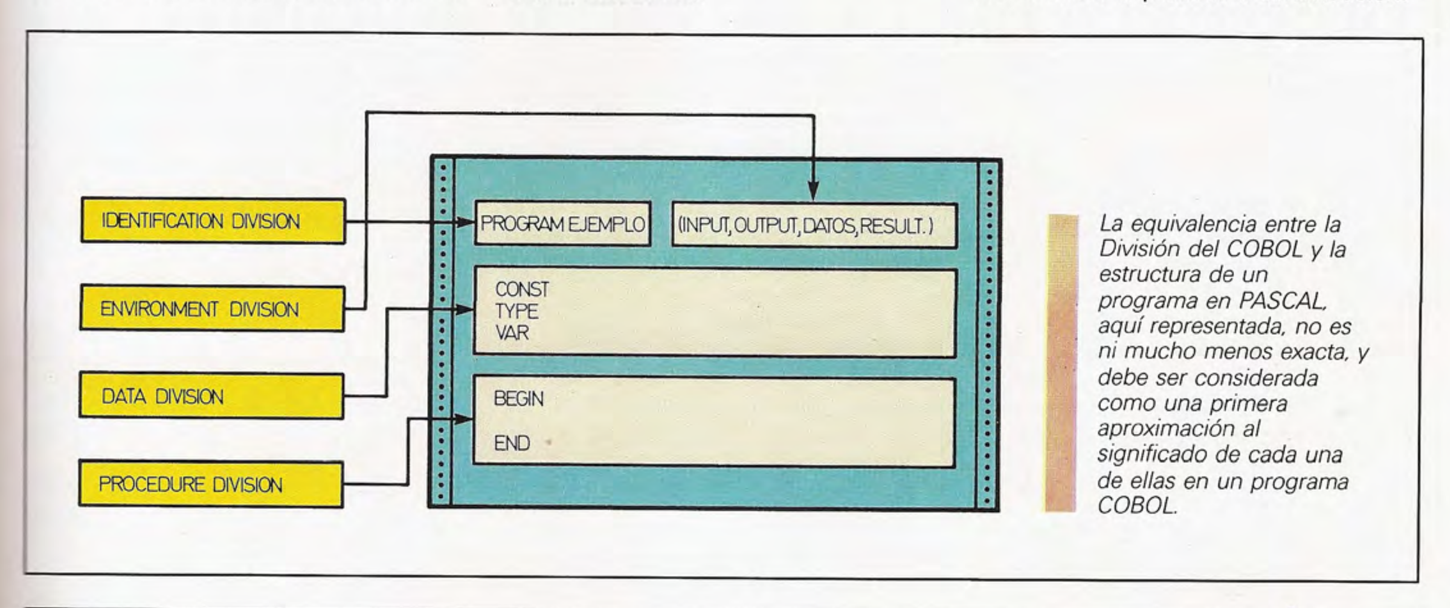
tura, escritura y actualización de registros en un fichero abierto), ACCEPT y DISPLAY (para la entrada y salida, respectivamente, de datos por un periférico en concreto, como pueda ser el teclado o la pantalla).

* *Instrucciones de manipulación de datos*

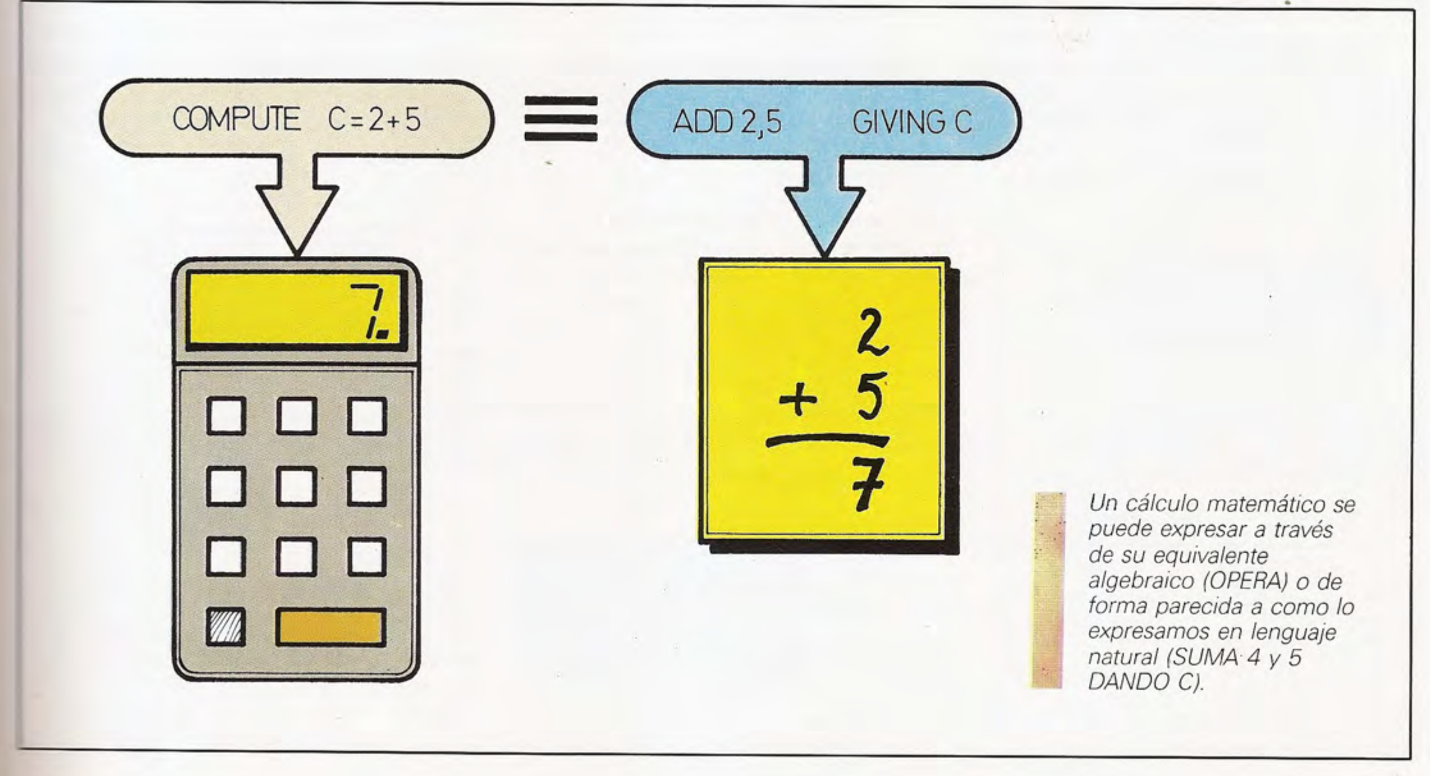
Por manipulación de datos deberemos entender el trabajo con los identificadores de dichos datos, sin tener en cuenta el valor numérico que tienen asignado. En este grupo se encuentran las instrucciones MOVE, que asigna a un identificador el contenido de otro, y EXAMINE, encargada de inspeccionar el contenido de un identificador y, en su caso, efectuar en él alguna transformación (por ejemplo, sustituir los espacios en blanco por asteriscos).

* *Instrucciones aritméticas*

En este apartado deberemos encuadrar las instrucciones que tratan los identificadores como datos numéricos, sobre los que pueden ejecutarse cualesquiera operaciones algebraicas. Estas instrucciones son ADD (suma), SUBSTRACT (resta), MULTIPLY (multiplica), DIVIDE (divide) y COMPUTE, encargada de obtener el resultado de la operación matemática expresada a continuación.



La equivalencia entre la División del COBOL y la estructura de un programa en PASCAL, aquí representada, no es ni mucho menos exacta, y debe ser considerada como una primera aproximación al significado de cada una de ellas en un programa COBOL.



Un cálculo matemático se puede expresar a través de su equivalente algebraico (OPERA) o de forma parecida a como lo expresamos en lenguaje natural (SUMA 4 y 5 DANDO C).

```
* ACCEPT identificador-1

ACCEPT identificador FROM { DATE
                            DAY
                            TIME }

ADD { identificador-1 } [ , identificador-2 ] ...
    { literal-1       } [ , literal-2       ]

    TO identificador-m [ROUNDED]

    [; ON SIZE ERROR instrucción-imperativa]

ADD { identificador-1 } , { identificador-2 } [ , identificador-3 ] ...
    { literal-1       }   { literal-2       } [ , literal-3       ]

    GIVING identificador-m [ROUNDED]

    [; ON SIZE ERROR instrucción-imperativa]

CLOSE nombre-de-archivo-1 [ REEL
                            UNIT
                            WITH LOCK ]

    [ , nombre-de-archivo-2 [ REEL
                              UNIT
                              WITH LOCK ] ] ...

COMPUTE identificador = expresión-calculable

DELETE nombre-de-archivo RECORD

DELETE nombre-de-archivo RECORD [; INVALID KEY instrucción-imperativa]

* DISPLAY { identificador-1 }
          { literal-1       }

DIVIDE { identificador-1 } INTO identificador-2 [ROUNDED]
       { literal-1       }

    [; ON SIZE ERROR frase-imperativa]

DIVIDE { identificador-1 } INTO { identificador-2 }
       { literal-1       }      { literal-2       }

    GIVING identificador-3 [ROUNDED] [REMAINDER identificador-4]

    [; ON SIZE ERROR frase-imperativa]

DIVIDE { identificador-1 } BY { identificador-2 }
       { literal-1       }    { literal-2       }

    GIVING identificador-3 [ROUNDED] [REMAINDER identificador-4]

    [; ON SIZE ERROR frase-imperativa]

EXIT

GO TO nombre-de-procedimiento

GO TO nombre-de-procedimiento-1 [, nombre-de-procedimiento-2]...

    , nombre-de-procedimiento-n DEPENDING ON identificador

                         { IS [NOT] GREATER THAN }
                         { IS [NOT] LESS THAN    }
IF { identificador-1  }  { IS [NOT] EQUAL TO     }  { identificador-2  }
   { literal-1        }  { IS [NOT] >            }  { literal-2        }
   { nombre-índice-1  }  { IS [NOT] <            }  { nombre-índice-2  }
                         { IS [NOT] =            }

    { frase-1; ELSE frase-2
      NEXT SENTENCE; ELSE frase-2
      frase-1; ELSE NEXT SENTENCE
      frase-1.                     }

IF identificador IS [NOT] { NUMERIC
                            ALPHABETIC }

    { frase-1; ELSE frase-2
      NEXT SENTENCE; ELSE frase-2
      frase-1; ELSE NEXT SENTENCE
      frase-1.                     }

IF identificador IS [NOT] { POSITIVE
                            NEGATIVE
                            ZERO     }

    { frase-1; ELSE frase-2
      NEXT SENTENCE; ELSE frase-2
      frase-1; ELSE NEXT SENTENCE
      frase-1.                     }

IF nombre-de-condición { frase-1; ELSE frase-2
                         NEXT SENTENCE; ELSE frase-2
                         frase-1; ELSE NEXT SENTENCE
                         frase-1.                     }

IF condición { [ AND ]  condición }
             { [ OR  ]            }

    { frase-1; ELSE frase-2
      NEXT SENTENCE;ELSE frase-2
      frase-1; ELSE NEXT SENTENCE
      frase-1.                     }

INSPECT identificader-1 TALLYING

    identificador-2 FOR { { FIRST   }  identificador-3 }
                        { { ALL     }  leteral-1       }
                        { { LEADING }                  }
                        { CHARACTERS                   }

    [ { BEFOR } INITIAL { identificador-4 } ]
      { AFTER }         { literal-2       }

INSPECT identificader-1 REPLACING

    { { ALL     }  identificador-5 }  BY  identificador-6
    { { LEADING }  literal-3       }      literal-4
    { { FIRST   }                  }
    { CHARACTERS                   }

    [ { BEFORE } INITIAL  identificador-7 ]
      { AFTER  }          literal-5

INSPECT identificador-1 TALLYING

    identificador-2 FOR { { ALL     } { identificador-3 } }
                        { { LEADING } { literal-1       } }
                        { CHARACTERS                      }

    [ { BEFORE } INITIAL { identificador-4 } ]
      { AFTER  }         { literal-2       }

    REPLACING

    { { ALL     } { identificador-5 } }  BY { identificador-6 }
    { { LEADING } { literal-3       } }     { literal-4       }
    { { FIRST   }                     }
    { CHARACTERS                      }

    [ { BEFORE } INITIAL { identificador-7 } ]
      { AFTER  }         { literal-5       }

MOVE { identificador-1 } TO identificador-2 [, identificador-3]...
     { literal         }

MULTIPLY { identificador-1 } BY identificador-2 [ROUNDED]
         { literal-1       }

    [; ON SIZE ERROR frase-imperativa]

MULTIPLY { identificador-1 } BY { identificador-2 }
         { literal-1       }    { literal-2       }

    GIVING identificador-3 [ROUNDED]

    [; ON SIZE ERROR frase-imperativa]

     { INPUT nombre-archivo-1 [, nombre-archivo-2] ...  }
OPEN { OUTPUT nombre-archivo-3 [, nombre-archivo-4] ... } ...
     { I-O nombre-archivo-5 [, nombre-archivo-6] ...    }
     { EXTEND nombre-archivo-7 [, nombre-archivo-8] ... }

PERFORM nombre-de-procedimiento-1 [ { THROUGH } nombre-procedimiento-2 ]
                                    { THRU    }

PERFORM nombre-de-procedimiento-1 [ { THROUGH } nombre-procedimiento-2 ]
                                    { THRU    }

    { identificador } TIMES
    { entero        }

PERFORM nombre-procedimiento-1 [ { THROUGH } nombre-procedimiento-2 ]
                                 { THRU    }

    UNTIL condición-1

PERFORM nombre-procedimiento-1 [ { THROUGH } nombre-procedimiento-2 ]
                                 { THRU    }

    VARYING { identificador-1   } FROM { identificador-2   }
            { nombre-de-índice  }      { nombre-de-índice-2 }
                                       { literal-1         }

    BY { identificador-3 } UNTIL condición-1
       { literal-2       }

READ nombre-de-archivo RECORD [INTO identificador]

    [; AT END frase-imperativa]

READ nombre-de-archivo RECORD [INTO identificador]

    [; INVALID KEY frase-imperativa]

READ nombre-de-archivo NEXT RECORD [INTO identificador]

    [; AT END frase-imperativa]

REWRITE nombre-de-registro [FROM identificador]

REWRITE nombre-de-registro [FROM identificador]

    [; INVALID KEY frase-imperativa]

SEARCH identificador-1 [ VARYING { identificador-2     } ]
                                 { nombre-de-índice-1  }

    [; AT END frase-imperativa-1]

    ; WHEN condición-1 { frase-imperativa-2 }
                       { NEXT SENTENCE      }

    [; WHEN condición-2 { frase-imperativa-3 } ] ...
                        { NEXT SENTENCE      }

SET nombre-de-índice-1 [, nombre-de-índice-2] ...

    TO { nombre-de-índice-3
         elemento-dato-índice
         identificador-1
         entero-1              }

SET nombre-de-índice-1 [, nombre-de-índice-2] ... { UP BY   }
                                                  { DOWN BY }

    { identificador-2 }
    { entero-2        }

SET elemento-dato-índice-1 [, elemento-dato-índice-2] ...

    TO { nombre-de-índice         }
       { elemento-dato-índice-3   }

SET identificador-1 [, identificador-2] ... TO nombre-de-índice

                                  { IS EQUAL TO          }
                                  { IS =                 }
START nombre-de-archivo [ KEY     { IS GREATER THAN      } elemento-dato ]
                                  { IS >                 }
                                  { IS NOT LESS THAN     }
                                  { IS NOT <             }

    [; INVALID KEY frase-imperativa]

STOP { RUN     }
     { literal }

STRING { identificador-1 } [ , identificador-2 ] ...
       { literal-1       } [ , literal-2       ]

    DELIMITED BY { identificador-3 } [ , { identificador-4 }
                 { literal-3       }     { literal-4       }
                 { SIZE            }

    [ , identificador-5 ] ... DELIMITED BY { identificador-6 } ] ...
    [ , literal-5       ]                  { literal-6       }
                                           { SIZE            }

    INTO identificador-7 [WITH POINTER identificador-8]

    [; ON OVERFLOW frase-imperativa]

SUBTRACT { identificador-1 } [ , identificador-2 ] ...
         { literal-1       } [ , literal-2       ]

    FROM identificador-m [ROUNDED] [; ON SIZE ERROR frase-imperativa]

SUBTRACT { identificador-1 } [ , identificador-2 ] ...
         { literal-1       } [ , literal-2       ]

    FROM { identificador } GIVING identificador-n [ROUNDED]
         { literal-m     }

    [; ON SIZE ERROR frase-imperativa]

UNSTRING identificador-1

    [ DELIMITED BY [ALL] { identificador-2 }
                         { literal-1       }

    [ , OR [ALL] { identificador-3 } ] ... ] INTO identificador-4
                 { literal-2       }

    [, DELIMITER IN identificador-5] [, COUNT IN identificador-6]

    [, identificador-7 [, DELIMITER IN identificador-8]

    [, COUNT IN identificador-9] ] ...

    [WITH POINTER identificador-10] [TALLYING IN identificador-11]

    [; ON OVERFLOW frase-imperativa]

                                                          { nombre-de-archivo }
USE AFTER STANDARD { EXCEPTION } PROCEDURE ON             { INPUT             }
                   { ERROR     }                          { OUTPUT            }
                                                          { I-O               }
                                                          { EXTEND            }

WRITE nombre-de-registro [FROM identificador-1]

    [ { BEFORE } ADVANCING { { identificador-2 } [ LINE  ] } ]
      { AFTER  }           { { entero          } [ LINES ] }
                           { { nombre-mnemotécnico }       }
                           { { PAGE                }       }

    [; AT { END-OF-PAGE } frase-imperativa ]
          { EOP         }

WRITE nombre-de-registro [FROM identificador]

    [; INVALID KEY frase-imperativa]
```

Formato de las instrucciones de la PROCEDURE DIVISION, encargada de contener el programa propiamente dicho.

** Instrucciones condicionales y de bifurcación*

En ocasiones es necesario tomar dentro del programa ciertas decisiones, o bien elegir un camino u otro según el resultado de una operación. Para estos casos están disponibles la instrucción condicional, IF (escoge entre dos opciones en función de que se cumpla o no una condición propuesta), y las distintas instrucciones de salto o bifurcación, que son GO TO (salta incondicionalmente al párrafo cuya etiqueta se menciona en la instrucción), GO TO DEPENDING ON (salto condicional a la etiqueta que mencione la instrucción, en función del valor de cierta variable mencionada), STOP (detención temporal o definitiva del programa), EXIT (punto de salida de un párrafo o proceso) y ALTER (cambia la referencia a una etiqueta o nombre de párrafo).

** Instrucciones de control*

Una estructura enormemente utilizada en todo tipo de programas es el bucle o lazo, consistente en la repetición de un determinado grupo de instrucciones un cierto número de veces, o bien hasta que se cumpla una condición. Para crear este tipo de estructuras se dispone en COBOL de la instrucción PERFORM.

Subprogramación

Una de las grandes ventajas de los programas de ordenadores es su portabilidad, esto es, pueden ser llevados de un lugar a otro y ejecutados por cuantas máquinas sea necesario, siempre dentro de una serie de restricciones impuestas por el hardware. Para poder aprovechar esta situación es conveniente desarrollar los programas en forma de módulos autocontenidos, lo que favorece su fácil intercambio y modificación. En general, siempre es más sencillo de entender y modificar un programa corto que realice una única tarea con dos o tres ficheros, que un programa de cientos de líneas encargado de la gestión completa de un gran volumen de datos. Estos módulos, denominados subprogramas, se integrarán más tarde en el programa principal, encargado de gestionar la entrada en funcionamiento y la detención de cada uno de ellos en el momento adecuado, entre otras cosas.

La instrucción empleada en COBOL para llamar a un subprograma es CALL (en inglés, llama), seguida del nombre asignado al programa (o subprograma) llamado. En caso de que sea necesario, la instrucción de llamada puede contener los identificadores del programa principal que pueden ser utilizados por el programa secundario. Previamente deberán haberse declarado en la Linkage Section (sección de enlace) de la Data Division del subprograma los nombres que el mencionado subprograma va a asignar a las variables del programa principal.

Todo subprograma debe, tras ser ejecutado, devolver el control al programa

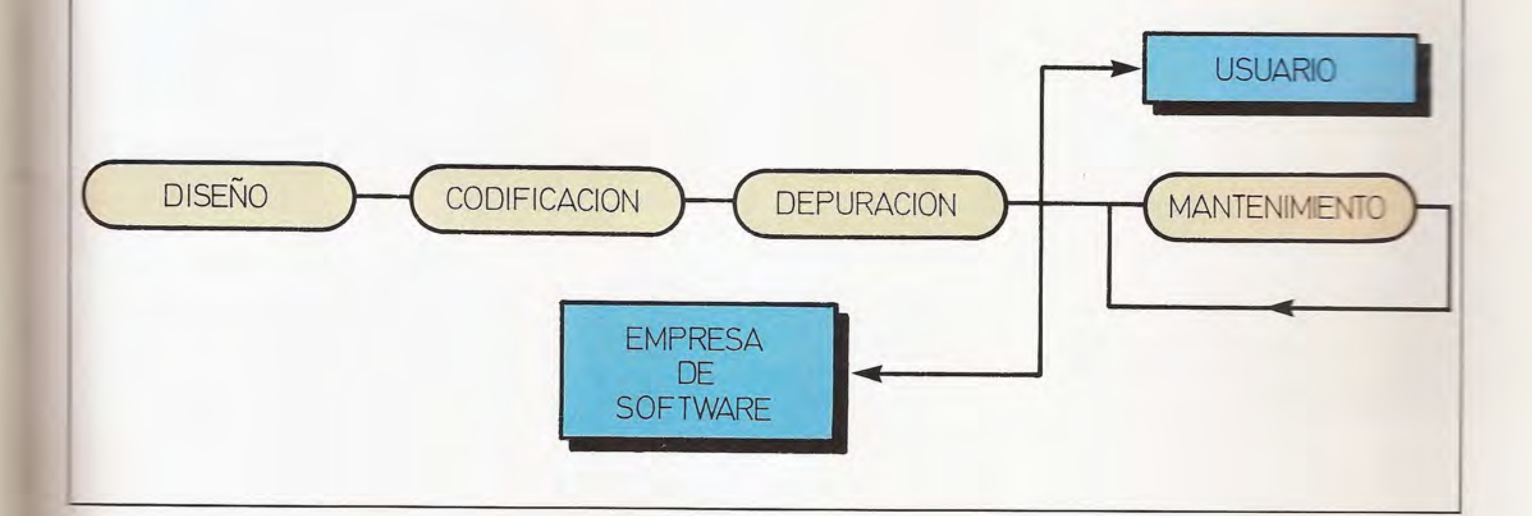
Ciclo de vida de un programa

Se ha mencionado que el hecho de especificar la estructura de datos y el programa en sí (el algoritmo a seguir) en dos partes diferenciadas del programa ayuda a la introducción de futuros cambios y a la corrección de errores.

El ciclo de vida de un programa comprende todas las fases que van desde el diseño de la estructura del mismo hasta que le es entregado al usuario, incluyendo la corrección de los posibles errores que se detecten una vez en funcionamiento.

La subsodicha partición del programa en dos zonas diferenciadas es útil prácticamente en todas las fases, pero es especialmente en la de mantenimiento. En esta última se corrigen los errores que el usuario encuentra en el funcionamiento del programa que se le entregó.

No hay que olvidar que un gran proyecto informático es un esfuerzo que pude ocupar a muchas personas durante un largo período de tiempo. Cuando aparecen los errores en la fase de mantenimiento, los componentes del equipo de programación pueden no ser los mismos, por lo que el hecho de que el propio programa se «autoexprese» a través de sus estructuras de datos y algoritmos resultará de gran ayuda.



llamante. Para ello se emplean las instrucciones EXIT PROGRAM o GOBACK.

Recapitulación y ejemplos

El nombre de una variable COBOL está compuesto por hasta treinta caracteres, entre letras, dígitos y guiones (estos últimos especialmente útiles en nombres de variables constituidos por más de una palabra). Al menos uno de estos caracateres ha de ser una letra, y el último no puede ser un guión. Nombres válidos serían, pues:

```
SALDO-TOTAL
CANTIDAD-EN-EXISTENCIA
8WIXZC
```

Lo que normalmente recibe el nombre de constantes en otros lenguajes se llama en COBOL literales, y pueden ser de dos tipos: numéricos y no numéricos, en cuyo caso van encerrados entre comillas simples. Ejemplos de literales pueden ser:

124.71 .... literal numérico
"Empleado" .... literal no numérico

También existen las constantes figuradas, que son, entre otras, ZERO, ZEROES, SPACE, SPACES y QUOTE. Si en la Data Division hemos definido la variable ABC como capaz de contener seis caracteres, al hacer:

```
MOVE SPACES TO ABC.
```

la variable ABC pasa a contener sólo espacios en blanco (seis). Observemos el punto con el que finaliza la sentencia. En COBOL, las sentencias van separadas entre sí por un punto y un espacio como mínimo.

Palabras reservadas del Cobol		
ACCEPT	FOR	READ
ADD	FROM	REEL
AFTER	GIVING	REMAINDER
ALL	GO TO	REPLACING
ALPHABETIC	GREATER	REWRITE
AND	IF	ROUNDED
BEFORE	INPUT	RUN
BY	INSPECT	SEARCH
CHARACTER	INTO	SENTENCE
CLOSE	INVALID	SET
COMPUTE	I-O	SIZE
COUNT	KEY	START
DATE	LEADING	STRING
DAY	LESS	SUBTRACT
DELETE	LOCK	TALLYNG
DELIMITED	MOVE	THROUGH
DEPENDING	MULTIPLY	THRU
DISPLAY	NEGATIVE	TIME
DIVIDE	NEXT	TIMES
DOWN	NOT	TO
ELSE	NUMERIC	UNIT
END	OPEN	UNSTRING
END-OF-PAGE	OR	UNTIL
EOP	OUTPUT	UP
EQUAL	OVERFLOW	USE
ERROR	PAGE	WARYING
EXCEPTION	PERFORM	WHEN
EXIT	POINTER	WRITE
EXTEND	POSITIVE	
FIRST	PROCEDURE	

Las operaciones aritméticas se pueden realizar dejando el resultado en uno de los operandos o especificando la variable que va a recibir el resultado. Por ejemplo,

```
ADD 1 TO CONTADOR.
```

incrementa la variable contador (resultado en uno de los operandos, en este caso CONTADOR),

```
ADD A,B TO C.
```

añade A y B a C, dejando el resultado en C (esto es, C=A+B+C), y

```
ADD A,B,C, GIVING D
```

realiza la misma operación, pero almacenando el resultado en la variable D.

De forma análoga se expresan restas (SUBSTRACT) multiplicaciones (MULTIPLY) y divisiones (DIVIDE). La sentencia COMPUTE ofrece una interesante manera de abreviar la escritura de los cálculos (ver figura).

Empleando la mencionada sentencia, tendremos las siguientes instrucciones, completamente equivalente a las anteriores:

```
COMPUTE CONTADOR=CONTADOR+1.
COMPUTE C=A+B+C.
COMPUTE D=A+B+C.
```

Las secciones en que el programador divide a la Procedure Division son una especie de subprogramas que pueden ser invocados a través de las sentencias PERFORM o GO TO.

La entrada/salida se realiza a través de las sentencias ACCEPT, DISPLAY, OPEN, READ, WRITE y CLOSE; de ellas, las cuatro últimas están dedicadas al manejo de ficheros. Por ejemplo, podemos especificar:

```
DISPLAY "HOLA, COMO ESTAS".
```

y aparecerá la frase HOLA, COMO ESTAS en el dispositivo de salida especificado en la Configuration Division. Como en todo literal no numérico, no pueden aparecer comillas. Para escribir la misma frase entrecomillada deberíamos emplear la constante figurada QUOTE:

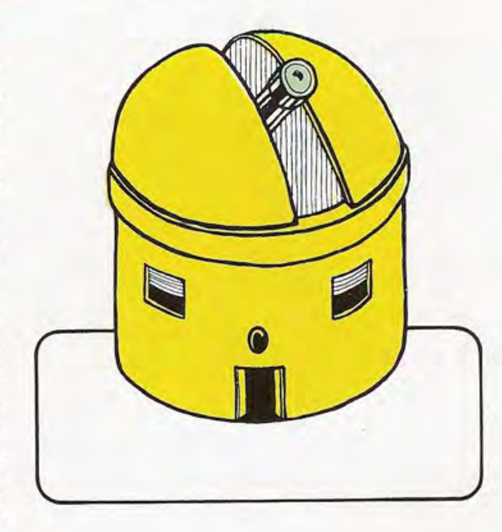
```
DISPLAY QUOTE "HOLA, COMO ESTAS" QUOTE.
```

FORTH (1)

Introducción al lenguaje FORTH

Todos los lenguajes de programación acogen una serie de características primordiales que los hacen más o menos idóneos para determinadas aplicaciones. Tales características derivan de la propia filosofía del lenguaje, y en menor grado de la presencia o ausencia de determinadas instrucciones dentro de su vocabulario.

El FORTH es un lenguaje que se caracteriza por permitir un gran aprovechamiento de todas las posibilidades y recursos de la máquina. Son muchas las facultades del FORTH que lo acercan a los lenguajes de bajo nivel; y ello, tratándose de un lenguaje de alto nivel, es ya decir mucho. Semejante propiedad hace que gran parte de sus instrucciones no sean todo lo potentes que cabría esperar del repertorio de un lenguaje de alto nivel; aunque, bien es cierto, que sus virtudes compensan con creces este inconveniente. No terminan aquí sus peculiaridades. La mayor parte de los lenguajes de alto nivel se suelen clasificar en compilados o interpretados; si bien, algunos (el BASIC, por ejemplo) admiten ambas posibilidades. El FORTH no se ajusta a esta clasificación. Su peculiar estructura no es ni compilada ni interpretada. Un programa FORTH se compone de una serie de rutinas y, a su vez, el mismo programa es una rutina más con la que es posible construir nuevos programas. Pues bien, estas rutinas, una vez creadas son compiladas, y quedan así en la memoria del ordenador, aunque el trabajo con el lenguaje se realiza en modo interpretado. En el ámbito del FORTH anidan tres o cuatro conceptos fundamentales que difieren de los usuales en otros lenguajes de programación. Estos conceptos no son especialmente difíciles, pero son nuevos, y familiarizarse con ellos puede llevar algún tiempo. Ante todo, hay que borrar de la mente ideas preconcebidas: en realidad, el FORTH es un lenguaje tan fácil como cualquier otro.



El FORTH nació como una herramienta de ayuda con la que formular los complicados cálculos necesarios en los observatorios astronómicos.

El concepto de pila

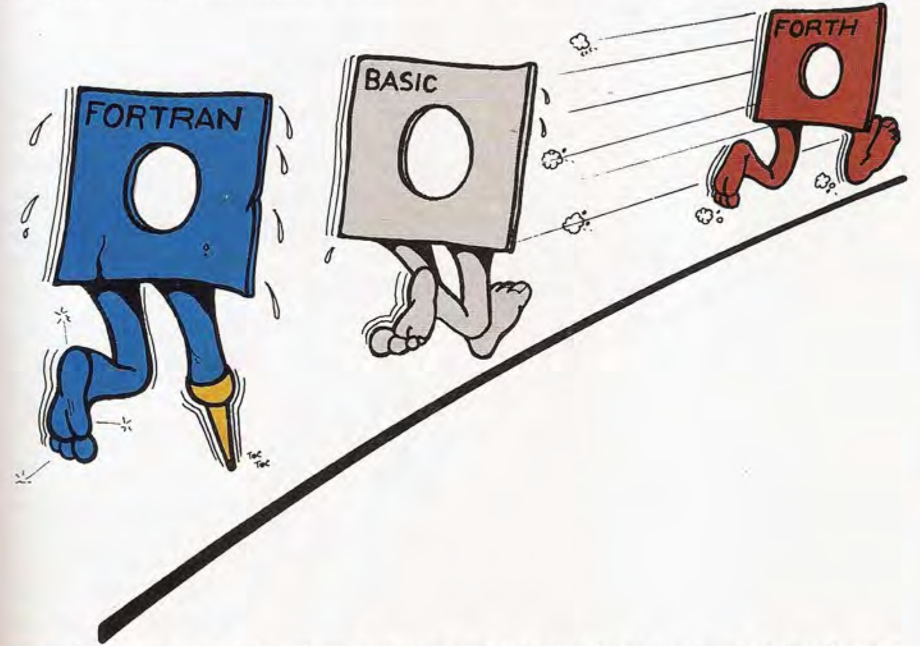
El primer concepto a definir es el de *pila*. Para enunciarlo, lo más cómodo es buscar un ejemplo en la vida real. Y no es difícil encontrarlo: por ejemplo, un apilamiento de platos fregados es una pila.

Observemos la forma de trabajar con uno de estos apilamientos. En primer lugar, una vez fregados, los platos se van depositando en la pila... ¿Pero en qué lugar de la pila? Resulta evidente que en la parte baja no se pueden colocar. Por tanto sólo es posible colocarlos en la cima.

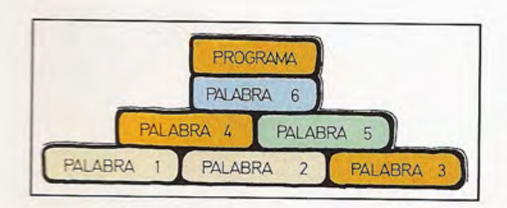
¿Y de dónde iremos recogiendo los platos a medida que los vayamos necesitando? Pues de la parte baja desde luego que no: la pila podría terminar en un verdadero estropicio. No cabe duda que habrá que irlos tomando, ordenadamente, de la zona superior, de la cima de la pila.

Este es, precisamente, el modo de trabajar con la pila. La pila es una zona de memoria en la que los datos se almacenan y se extraen de la misma forma que los platos de nuestro ejemplo. La estructura de almacenamiento de la pila es conocida con el nombre de *lista LIFO* (last input first output); o lo que es lo mismo, pero en castellano: el último en entrar es el primero en salir. Esta estructura condiciona la forma de trabajo con este lenguaje.

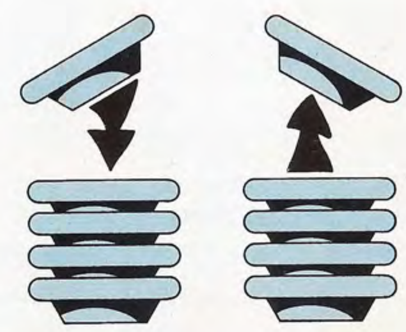
Realmente, el FORTH da entrada a dos pilas: en una de ellas se almacenan los datos con los que vamos a trabajar, se trata de la pila normal; la segunda es la llamada pila de retorno, y se emplea para almacenar valores de índices y va-



El FORTH es un lenguaje que destaca por su celeridad: tanto por la velocidad de ejecución de los programas como por la rapidez con la que es posible desarrollar software, partiendo de otros programas y rutinas ya existentes.



El FORTH es un lenguaje fundamentado en la creación de palabras que, posteriormente, pueden ser empleadas para crear nuevas palabras más evolucionadas.



Secuencia de entrada y salida de datos en la pila.

Un callejón sin salida es también un ejemplo ilustrativo de pila: el último coche en entrar ha de ser el primero en salir.

riables de bucles. Desde luego, además de estos dos, hay que contar con la habitual pila del sistema.

Palabras

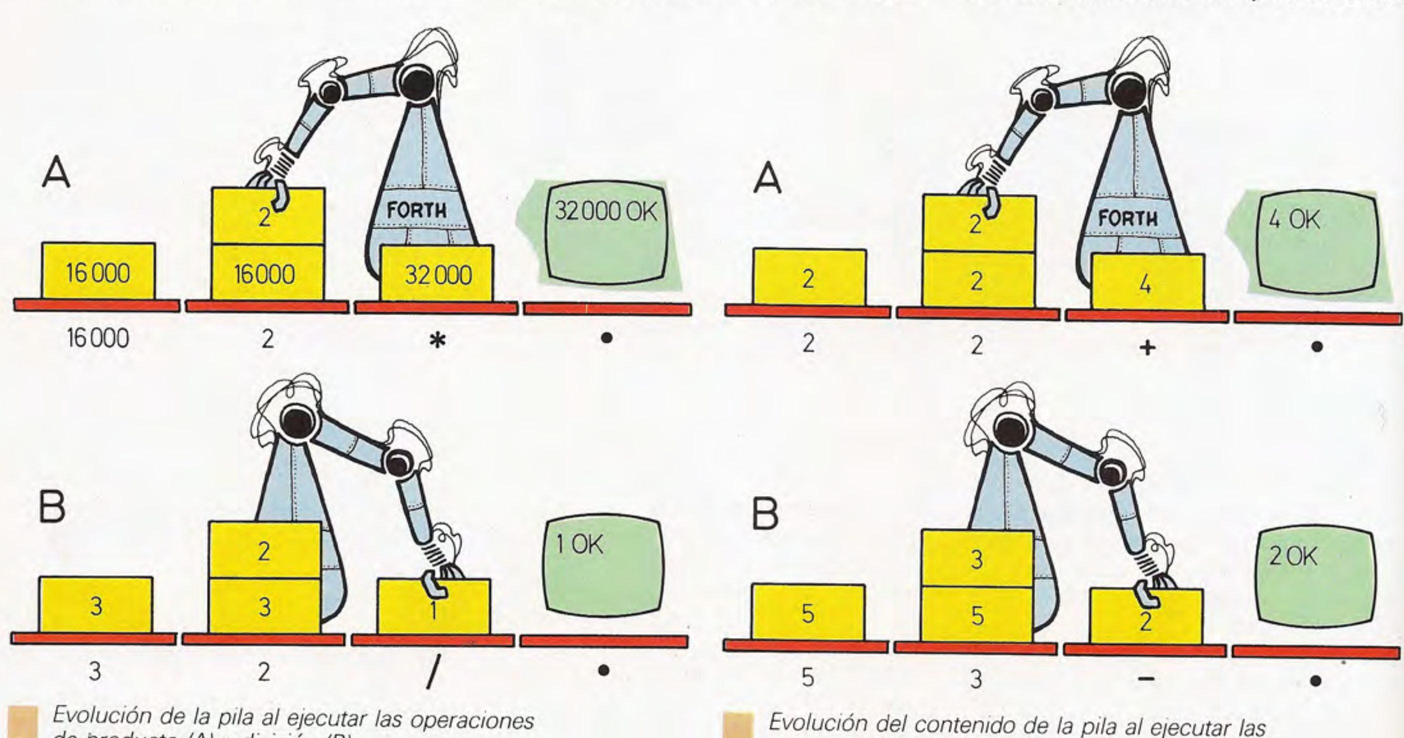
La palabra es uno de los conceptos básicos del FORTH y tal vez el más importante. Del mismo modo que en el lenguaje corriente bautizamos con una palabra a las cosas que todavía no tienen un nombre, en FORTH podemos bautizar a una secuencia de acciones o instrucciones por medio de una palabra.

De esta forma, al emplear una palabra, lo que haremos es referirnos al nombre con el que hemos bautizado a una serie de acciones. De ello se deduce que una palaba es un ente abstracto que asocia una serie de acciones o «definición» a un nombre. Una palabra FORTH consiste en un conjunto de caracteres delimitados por espacios. Dada la importancia que tienen en el FORTH los espacios, en su función de separadores, una palabra no puede contener ningún espacio en su interior. En principio, las palabras pueden ser de cualquier longitud, pero sólo los primeros caracteres serán significativos (por lo general, los 31 primeros).

De lo explicado hasta ahora, se puede deducir fácilmente que un programa FORTH es extraordinariamente modular. En efecto, la confección de un programa consiste en ir construyendo nuevas estructuras basadas en estructuras antiguas ya existentes. A esto hay que unir el hecho de que estructuras imprescindibles en otros lenguajes, como el GOTO, carecen aquí de sentido, con las consiguientes ventajas.

El diccionario

Cuando se creó el lenguaje FORTH, se pretendía garantizar el rápido acceso a



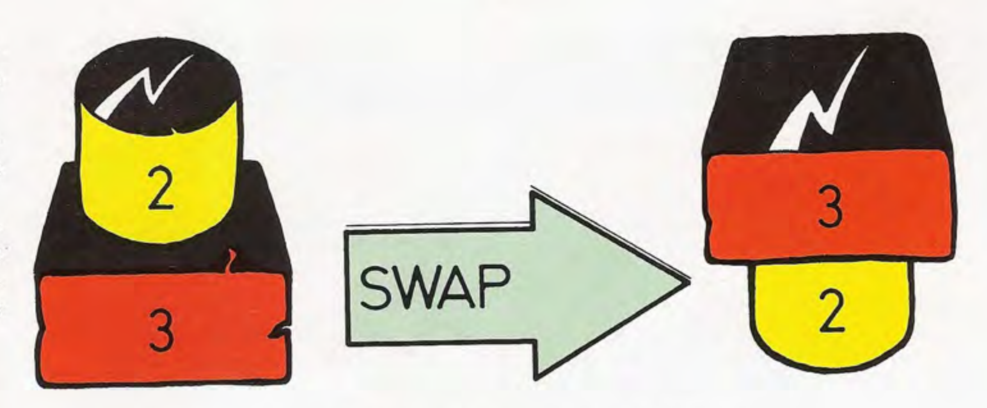
Evolución de la pila al ejecutar las operaciones de producto (A) y división (B).

Evolución del contenido de la pila al ejecutar las operaciones de suma (A) y resta (B).

un catálogo de funciones y poder definir otras nuevas funciones a partir de las ya existentes.

Recordemos que la palabra constituye el elemento esencial del FORTH e implica una acción determinada.

El conjunto de palabras FORTH conforma el diccionario. Un diccionario que puede enriquecerse mediante la creación de nuevas palabras que sumar a las ya existentes.



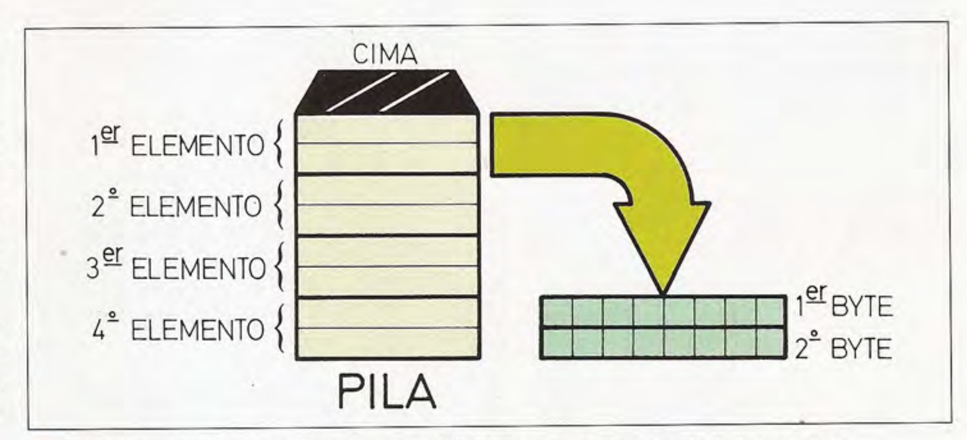
SWAP es una palabra FORTH integrada en el grupo de órdenes para la manipulación de la pila. Concretamente, SWAP intercambia los dos números que ocupan las posiciones superiores de la pila.

Números

El ordenador, al analizar lo que haya introducido el usuario, intentará, en primer lugar, asimilarlo como una palabra; si no lo consigue, entenderá que se trata de un número.

Los números pueden expresarse en cualquier base, de la 2 a la 36. En todo caso, al proceder a su conexión, el equipo trabajará en decimal, el sistema de numeración más común; no obstante, en cualquier momento puede ordenarse un cambio de base.

Los números pueden ser de diversos tipos; por ejemplo, enteros de simple o doble precisión. Los de precisión simple sólo pueden llegar al valor 32767 para positivos y a 32768 para negativos.



En la mayoría de las versiones de FORTH, cada elemento de la pila consta de un total de 16 bits. Ello significa que la pila utiliza realmente dos posiciones de memoria para el almacenamiento de cada dato.

Notación polaca inversa

Esta notación representa otra de las particularidades del FORTH. Consiste en suministrar el operador aritmético tras los datos sobre los cuales habrá de operar. Por ejemplo, para sumar dos números, la operación sería la siguiente:

```
2 2 +
```

La explicación es muy sencilla si pensamos en la «pila». Los datos son tomados y almacenados en la pila y, a continuación cuando el procesador observa la palabra «+» depositada en el punto superior de la pila, realiza la acción de tomar los dos primeros números que encuentre en la pila (retirándolos de la misma), los suma y coloca el resultado en la cima de la pila.

Aritmética

Como ya se ha indicado, los operandos han de estar depositados en la pila antes de que se realice la operación. Por ejemplo:

```
5 27 + . <CR>
```

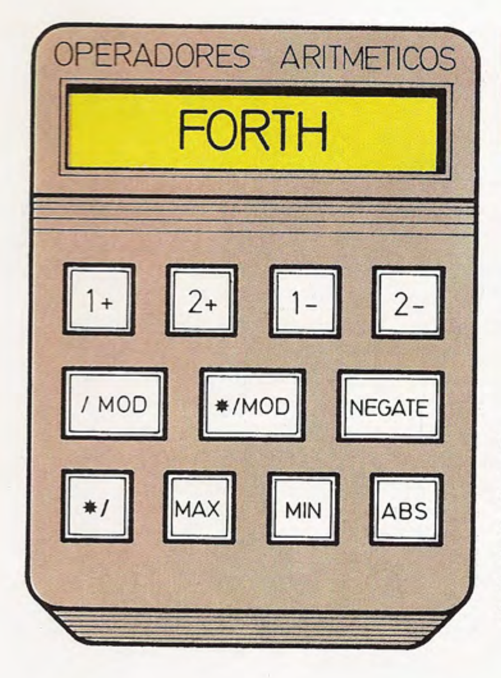
5: coloca cinco en la pila.
27: pone el valor veintisiete en la pila.
+: suma los dos números de la parte superior de la pila y los elimina de la misma, depositando el resultado de la operación en la cima de la pila.
.: es la palabra FORTH que se emplea para escribir en la pantalla el valor que se encuentra en la cima de la pila.

Aspecto de un programa FORTH

Un punto muy importante en todo programa FORTH es la correcta colocación de los espacios en blanco. Los espacios en blanco son los separadores más importantes. De la adecuada colocación de estos espacios depende la interpretación de las palabras introducidas. De cualquier forma, el editor se encargará normalmente de advertirnos del uso de palabras no existentes en su diccionario.

Operadores aritméticos

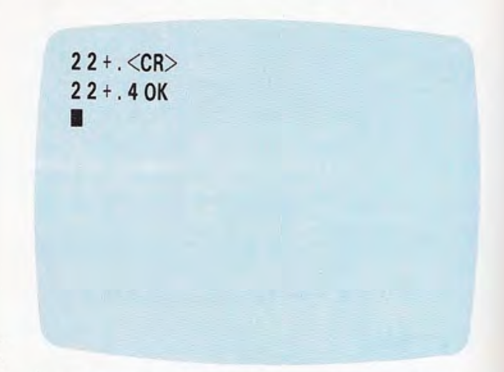
Los operadores aritméticos del FORTH también pertenecen a la categoría de las



El FORTH es un lenguaje de programación con un amplio repertorio de palabras cuya finalidad es la de actuar a modo de operadores aritméticos.

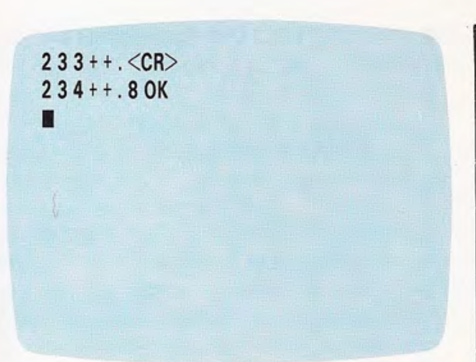
palabras. Se caracterizan por su peculiar forma de trabajo con la pila; de ahí que una parte importante de su explicación haya de versar sobre la manipulación de la pila. Veamos cuáles son los operadores aritméticos más significativos del FORTH:

+ Obtiene la suma de los elementos que lo siguen en la pila, y deposita el resultado en la cima de la misma.

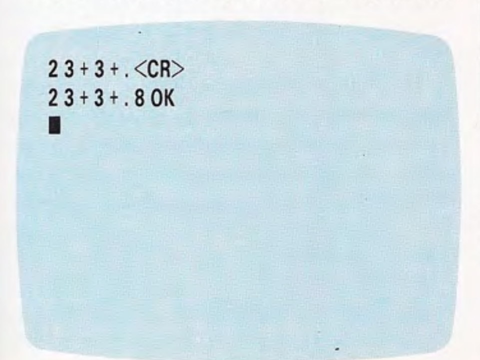


Estas operaciones se pueden encadenar, y para hacerlo hay que tener en cuenta el modo de trabajo de la pila. Veamos, por ejemplo, la forma de realizar dos sumas asociadas de distinta forma:

Evolución de una pila al ejecutar determinadas operaciones matemáticas.

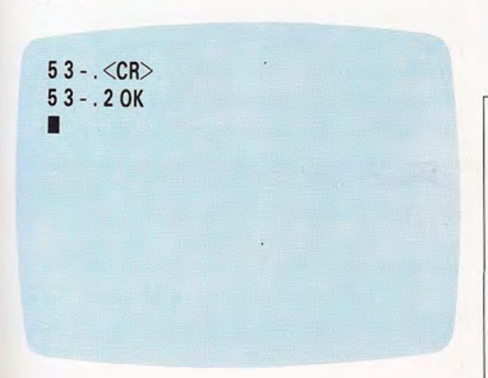


Operación que equivale a la siguiente

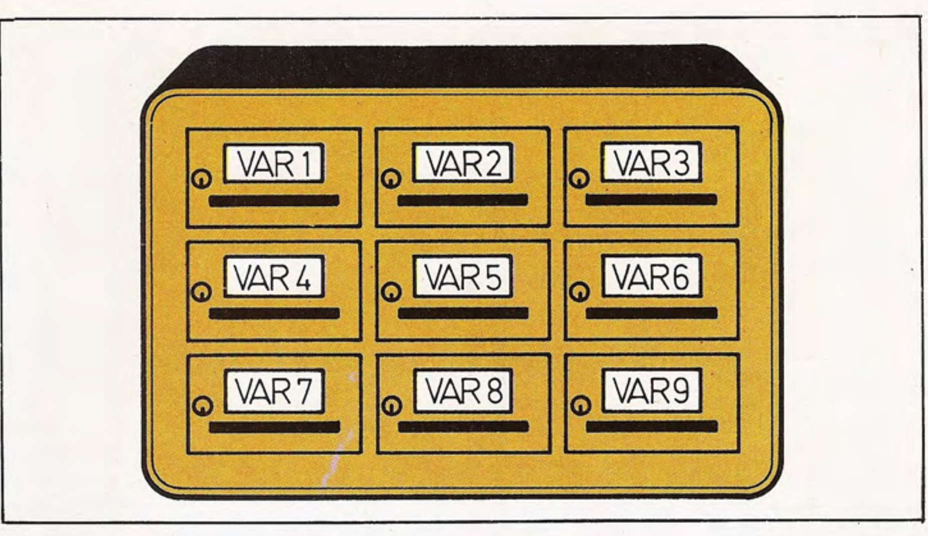


NOTA: En FORTH, el punto (.) sirve para mostrar por pantalla el contenido de la cima de la pila. El dato visualizado desaparece de la cima de la pila.

— Resta el segundo número introducido en la pila del introducido en primer lugar.

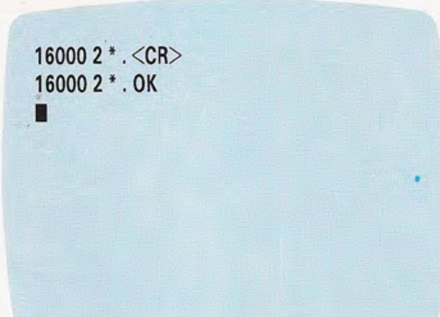


Al ejecutar la orden ROLL, el ordenador toma el número «n» situado en la cima de la pila (13 en nuestro caso); acto seguido, duplica el número situado «n» posiciones a partir de la cima de la pila y deposita la copia en la parte superior de la misma.

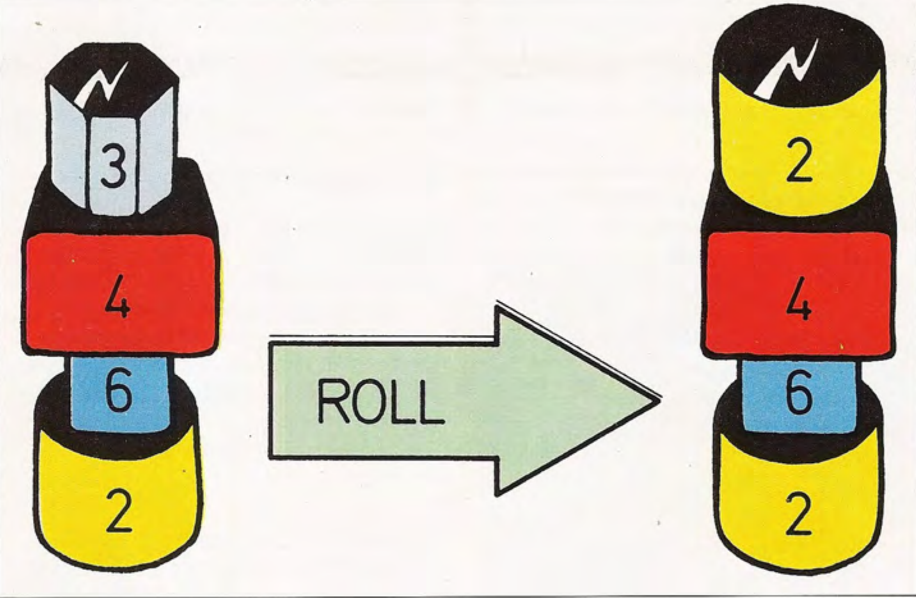


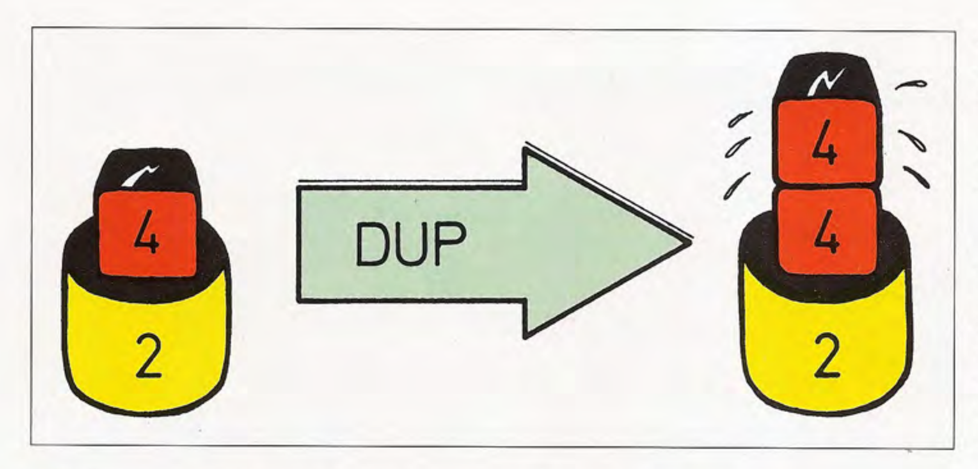
Cabe imaginar a la memoria como un conjunto de buzones, cada uno de ellos constituido por dos celdas de memoria (bytes), en donde es posible almacenar datos. Mediante la definición de constantes y variables asociamos un identificador a cada uno de los buzones, lo cual permitirá un acceso más cómodo a los mismos.

* Multiplica los dos números situados en las posiciones superiores de la pila. Deposita en la cima de la pila el resultado de la multiplicación.



/ Obtiene la división de los dos números de la cima de la pila, considerando como divisor al más cercano a la cima (el último que se introdujo) y al



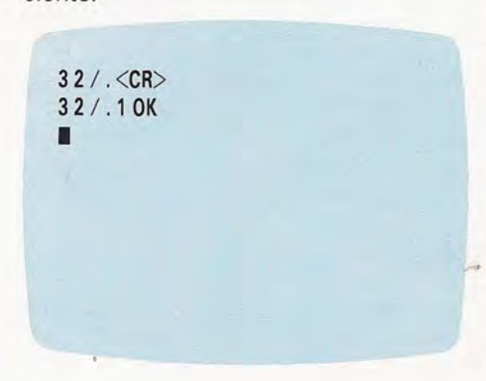


La palabra DUP duplica el dato localizado en la cima de la pila, siempre y cuando su valor no sea cero.



En nuestros días existen ya versiones del lenguaje FORTH adaptadas a múltiples microordenadores, incluso a equipos domésticos como es el caso del ZX-Spectrum.

otro como dividendo. El resultado de la operación sólo incluye el módulo del cociente.



Más operadores aritméticos

El lenguaje FORTH dispone de dos palabras destinadas a realizar incrementos; éstas son las dos siguientes:

1+ Suma una unidad al elemento situado en la cima de la pila, depositando en la cima el resultado de la operación.
Ejemplo:

```
2 1+ . <CR>
2 1+ . 3 OK
```

2+ Incrementa en dos unidades el elemento situado en la cima de la pila, dejando en su lugar el resultado de la operación.
Ejemplo:

```
23 2+ . <CR>
23 2+ . 25 OK
```

Los dos caracteres que componen estas palabras FORTH han de escribirse seguidos, sin dejar espacio en blanco entre ellos. De no hacerlo así, el resultado de la operación sería el mismo, si bien no coincidiría el proceso ejecutado y ello puede influir en el desarrollo del programa además de hacerlo más lento.

1- Esta nueva palabra permite restar una unidad al número situado en la cima de la pila, colocando en su lugar el resultado de la operación.
Ejemplo:

```
23 1- . <CR>
23 1- . 22 OK
```

2- Su funcionamiento es semejante al de la palabra anterior, pero en lugar de restar una sola unidad resta dos.
Ejemplo:

```
34 2- . <CR>
34 2- . 32 OK
```

/MOD Divide el primer número introducido en la pila por el segundo, depositando en ella dos números resultantes: el de la cima, coincidente con el módulo de la división, y el siguiente que corresponde al resto de la división.
Ejemplo:

```
3 6 /MOD . . <CR>
3 6 /MOD . 0 . 3 OK
```

Operadores evolucionados

El lenguaje FORTH posee, además, una serie de operadores complejos que ejecutan más de una operación, empleando para ello dos o incluso más operandos.

***/** Esta palabra multiplica el segundo número situado en la pila por el tercero, dividiendo el resultado por el situado en el primer lugar de la pila, esto es, en la cima.
Ejemplo:



La presencia del FORTH en el ámbito de los microordenadores es notable. Hace algún tiempo, incluso podía adquirirse un económico equipo doméstico —denominado Jupiter Ace— cuyo lenguaje residente era precisamente el FORTH, en lugar del tradicional BASIC.

```
2 4 5 */ . <CR>
2 4 5 */ . 1 OK
```

***/MOD** Ejecuta las dos operaciones descritas en el caso anterior, con la diferencia de que, además, proporciona el resto de la división.
Ejemplo:

```
2 4 5 */MOD . . <CR>
2 4 5 */MOD . 1. 3 OK
```

Las siguientes palabras FORTH realizan algunas manipulaciones de datos íntimamente relacionadas con operaciones aritméticas.

ABS Dicha palabra calcula el valor absoluto del número que se encuentra en la parte superior de la pila, y coloca el resultado en esa misma posición.
Ejemplo:

```
-2 ABS . <CR>
-2 ABS . 2 OK
```

MAX Toma los dos números emplazados en las posiciones superiores de la pila, y deposita el mayor de ambos en la cima.
Ejemplo:

```
2 9 MAX . <CR>
2 9 MAX . 9 OK

5 3 8 MAX MAX . <CR>
5 3 8 MAX MAX . 8 OK
```

MIN Muy semejante al anterior; toma los dos elementos superiores de la pila y coloca el menor de ellos en la cima.

NEGATE Cambia el signo del número situado en la cima de la pila.
Ejemplo:

```
3 NEGATE . <CR>
3 NEGATE . -3 OK
```

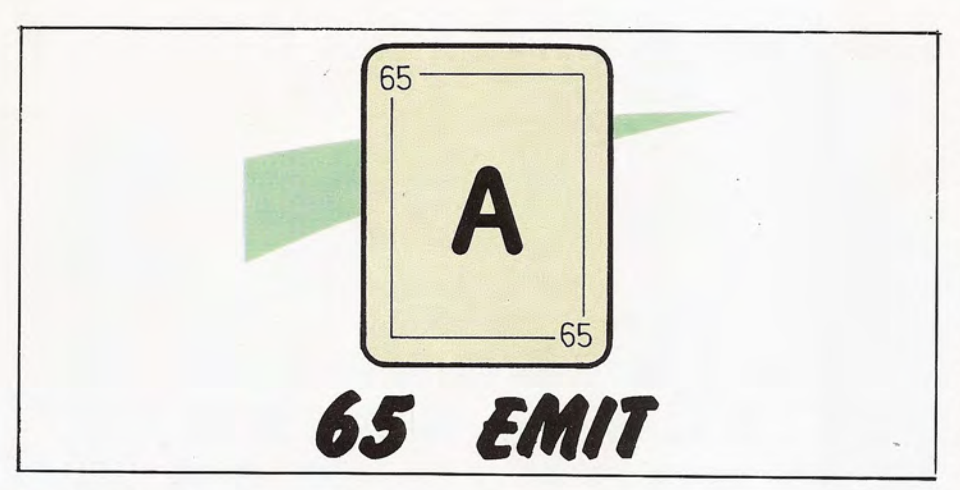
Cabe observar que en algunos dialectos FORTH no existe la función NEGATE; si bien, existen otras funciones que realizan el mismo cometido, entre ellas se encuentra MINUS.

En general todas estas funciones son habituales en cualquier dialecto FORTH; hay que constatar que éste es un lenguaje muy uniforme. Por lo demás, existe la ventaja de que si nuestra versión de este lenguaje no posee una determinada palabra, es muy sencillo crearla, con lo cual el problema desaparece.

El trabajo con la pila

Una vez estudiadas las palabras FORTH que permiten la realización de operaciones aritméticas, pasamos a describir una serie de palabras FORTH especializadas en realizar diferentes manipulaciones de la pila.

Este nuevo tipo de instrucciones es fundamental en el lenguaje FORTH; no hay que perder de vista, como hemos indicado antaeriormente, que la pila constituye la base de trabajo para todos los cálculos, ya que de ella se extraen los operandos. Con una de estas nuevas palabras hemos trabajado ya anteriormente; se trata de la palabra «.» que permite visualizar en la pantalla el número situado en la cima de la pila.



La palabra EMIT toma el elemento situado en la cima de la pila y visualiza en la pantalla el carácter cuyo código ASCII corresponde con dicho número.

DROP Elimina el número que se encuentra encima de la pila, con lo cual éste ya no se tiene en cuenta.
Ejemplo:

```
2 3 5 DROP . . <CR>
2 3 5 DROP . 3 . 2 OK
```

SWAP Intercambia los dos números situados en las posiciones superiores de la pila.
Ejemplo:

```
3 5 SWAP . . <CR>
3 5 SWAP . 3 . 5 OK
```

OVER Obtiene una copia del elemento situado en la segunda posición a partir de la cima de la pila, y coloca este duplicado en la cima de la pila. Como resultado, la pila mostrará en la cima un número igual al tercer elemento de la misma.
Ejemplo:

```
2 3 OVER . . . <CR>
2 3 OVER .2 .3 .2 OK
```

ROT Ejecuta una rotación de los tres elementos más próximos a la cima de la pila: coloca al tercer elemento en la cima y desplaza a los otros dos una posición hacia abajo.
Ejemplo:

```
1 2 3 ROT . . . <CR>
1 2 3 ROT .1 .3 .2 OK
```

```
1 2 3 ROT ROT ROT . . . <CR>
1 2 3 ROT ROT ROT .3 .2 .1 OK
```

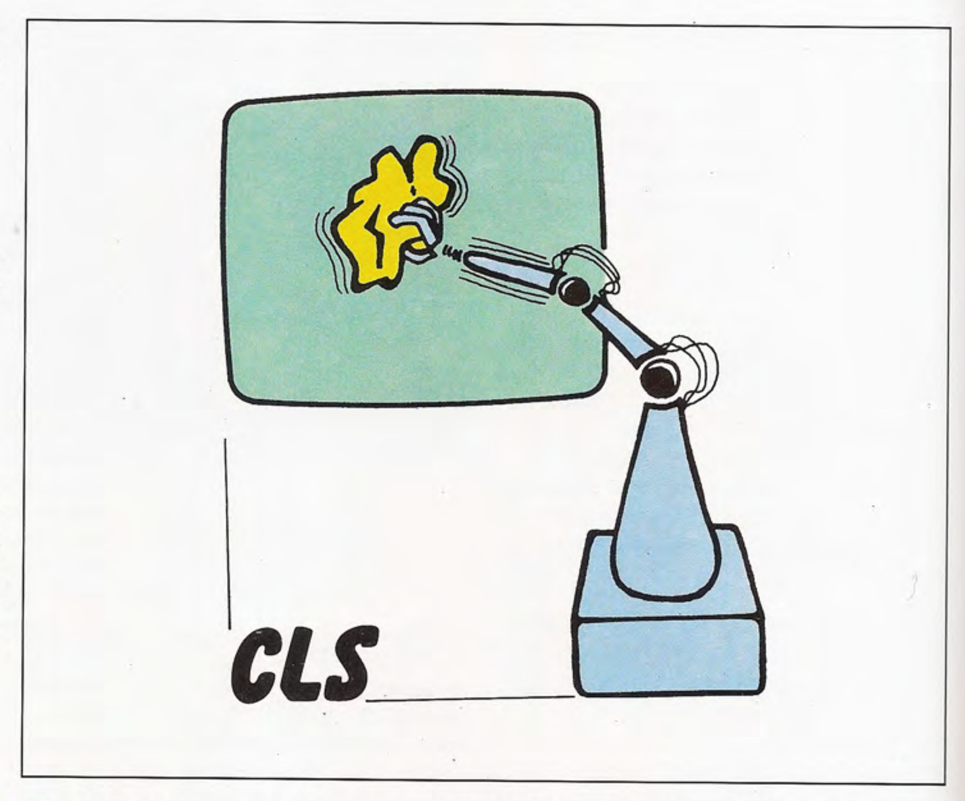
ROLL Toma el número n situado en la cima de la pila y lo utiliza como referencia para coger el número situado n posiciones a partir de la cima de la pila y duplicarlo, depositando una copia en la posición superior de la pila.
Ejemplo:

```
1 2 3 3 ROLL . <CR>
1 2 3 3 ROLL . 1 OK
```

DUP Duplica la cima de la pila en el caso de que ésta no sea cero.
Ejemplo:

```
3 DUP . . <CR>
3 DUP .3 .3 OK
```

R Copia la cima de la pila de retorno en la pila de operación.



CLS es el comando FORTH especializado en el borrado de la pantalla.

Constantes y variables

El lenguaje FORTH también permite definir constantes. Este concepto puede

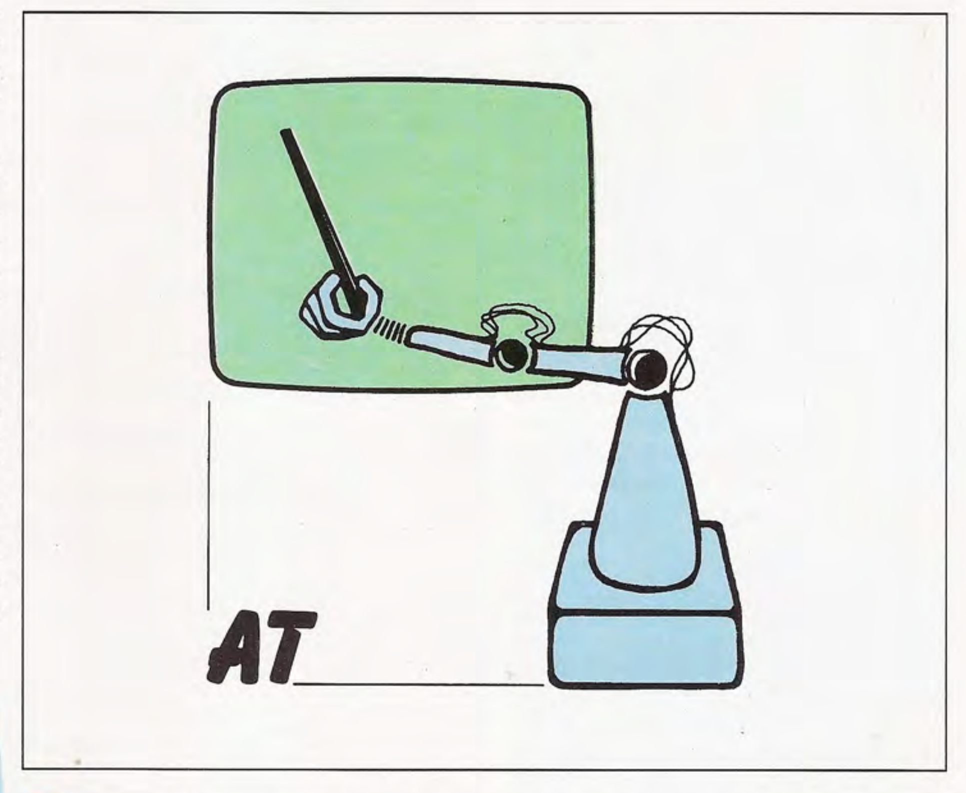
parecer un tanto extraño, especialmente para las personas acostumbradas a trabajar exclusivamente en BASIC, ya que tendrán la tendencia a asociar estas constantes con las variables.

En esencia, este artificio permite otorgar un nombre a un valor específico, valor que no puede ser cambiado a lo largo del proceso. Tal posibilidad hace posible asignar un nombre a un número, lo que facilitará las referencias al mismo dentro del programa.

Para realizar una definición de esta índole, hay que empezar trasladando el número que se pretende definir a la cima de la pila; acto seguido hay que indicar al ordenador que se desea definir una constante utilizando para ello la palabra CONSTANT seguida por el nombre que se desea otorgar a la constante. Veamos un ejemplo ilustrativo, tanto de definición de constantes como de su uso práctico.

```
23 CONSTANT EDAD <CR>
56 CONSTANT BAR <CR>

23 EDAD + . <CR>
23 EDAD + . 46 OK
56 BAR _ . <CR>
56 BAR _ . Ø OK
█
```



La orden AT tiene encomendada la tarea de precisar el punto de la pantalla en el que se realizará la próxima visualización de datos.

Al igual que ocurre con cualquier otro lenguaje informático, también en el FORTH es posible definir variables. Como ya conocerá todo aquel que se encuentre familiarizado con los temas de programación, una variable corresponde sencillamente a una posición de memoria cuyo valor se puede alterar a voluntad. El mecanismo para crear una variable FORTH pasa por utilizar la palabra VARIABLE. Para proceder a su definición, hay que empezar situando el valor inicial de la variable en la pila. A continuación, hay que indicar al ordenador que deseamos crear una variable, mediante el empleo de la palabra FORTH VARIABLE y, por último, hay que especificar el nombre que deseamos emplear como distintivo de dicha variable. Veamos algún ejemplo de definición de variables:

```
23 VARIABLE EDAD <CR>
23 VARIABLE EDAD OK

45 VARIABLE BASE <CR>
45 VARIABLE BASE OK
█
```

El siguiente paso dentro del proceso de trabajo con una variable, se concreta en la forma de trasladar a la pila el valor de la misma. Para ello, es necesario utilizar la palabra FORTH «@», seguida por el nombre identificador de la variable. Por ejemplo:

```
EDAD @ . <CR>
EDAD @ . 23
```

Una vez que el valor de la variable se encuentra depositado en la pila, es posible ya efectuar operaciones con el mismo.

La actuación de la palabra @ se resume en lo siguiente: busca en la pila una dirección de memoria, y recoge el contenido de dicha posición de memoria y de la siguiente, depositándolo en la pila. Cabe deducir, por lo tanto, que hace el uso de una palabra como identificador de una variable, equivale a asociar a dicha palabra un número representativo de una dirección de memoria. Veamos un ejemplo:

```
EDAD . <CR>
EDAD . 15452 OK
```

Como se observa, tras indicar el nombre de una variable, el ordenador muestra un número que se encuentra en la pila; número que coincide realmente con una dirección de memoria. Veamos qué sucede si colocamos directamente en la pila la referida dirección de memoria y empleamos la palabra @.

```
15452 @ . <CR>
15452 @ . 23 OK
```

En efecto, la pantalla muestra el valor almacenado en dicha posición. Ello significa que puede utilizarse la palabra @ para examinar el contenido de dos posiciones de memoria consecutivas; las que desee el programador.

Para cambiar el valor de una variable, hay que recurrir a la palabra «!», palabra que, a su vez, también es posible utilizarla para alterar el contenido de una dirección de memoria. Por ejemplo:

```
29 EDAD ! EDAD @ . <CR>
29 EDAD ! EDAD @ . 29 OK

29 15452 @ EDAD @ . <CR>
29 15452 @ EDAD @ . 29 OK
```

Otra posibilidad es la de modificar el valor de una constante, redefiniéndola mediante el empleo de la palabra asociada a la misma, aunque con la desventaja de que dicha palabra no podrá ser utilizada dentro de la definición de otra nueva palabra. Por ejemplo:

```
23 CONSTANT BALA <CR>
23 CONSTANT BALA OK

34 CONSTANT BALA REDEFINE BALA
   BALA . <CR>
34 CONSTANT BALA REDEFINE BALA
   BALA . 34 OK
```

Imprimiendo en pantalla

El lenguaje FORTH ofrece varios métodos para visualizar en pantalla la información puesta en juego. La forma más frecuente y sencilla consiste sencillamente, en el uso de la palabra FORTH «.». Esta palabra (.) presenta en pantalla el número que se encuentre, en ese preciso instante, en la cima de la pila. Por supuesto, si la pila está vacía la máquina responderá con un mensaje de error.

Para imprimir mensajes en pantalla podemos recurrir a la palabra FORTH .". Esta hará que se visualice en la pantalla el mensaje que se escriba a continuación, hasta que el ordenador encuentre otras comillas.

En determinadas versiones de FORTH, la palabra que nos ocupa (.") sólo se puede utilizar dentro de la definición de otra palabra FORTH. Suponiendo que la versión con la que se trabaja admite su ejecución en modo inmediato, la actuación de dicha palabra es la que ilustran los siguientes ejemplos:

```
." HOLA" <CR>
." HOLA" HOLA OK
```

```
." ESTO ES UN MENSAJE" <CR>
." ESTO ES UN MENSAJE" ESTO ES UN MENSAJE OK
```

No sólo es posible visualizar en pantalla un mensaje entrecomillado, sino que también puede escribirse en la misma un carácter dado por su código ASCII. Para ello, el lenguaje FORTH cuenta con la palabra clave EMIT. Al ejecu-

Tabla de órdenes FORTH

PALABRA	DESCRIPCION	TIPO
1+	Suma uno al elemento situado en la cima de la pila.	Instrucción aritmética.
2+	Suma dos al elemento situado en la cima de la pila.	Instrucción aritmética.
1-	Resta uno al elemento situado en la cima de la pila.	Instrucción aritmética.
2-	Resta dos al elemento situado en la cima de la pila.	Instrucción aritmética.
/MOD	Divide dos números y da el cociente y el resto.	Instrucción aritmética.
*/	Divide y multiplica.	Instrucción aritmética.
*/MOD	Divide y multiplica dando además el resto.	Instrucción aritmética.
ABS	Entrega el valor absoluto del número situado en la cima de la pila.	Instrucción aritmética.
MAX	Deposita en la cima de la pila el mayor de los dos números situados más cerca de la cima.	Instrucción aritmética.
MIN	Deposita en la cima de la pila el menor de los dos números situados más cerca de la cima.	Instrucción aritmética.
NEGATE	Cambia el signo del número situado en la cima de la pila	Instrucción aritmética.
DROP	Elimina el número situado en la cima de la pila.	Manipulación de la pila.
SWAP	Intercambia los dos elementos situados más cerca de la cima de la pila.	Manipulación de la pila.
OVER	Duplica el segundo elemento desde la cima de la pila y coloca este duplicado en la cima.	Manipulación de la pila.
ROT	Ejecuta una rotación de los tres elementos situados más cerca de la cima.	Manipulación de la pila.
ROLL	Toma un número N de la cima de la pila y busca el elemento situado en la posición enésima de la misma.	Manipulación de la pila.
DUP	Duplica la cima de la pila.	Manipulación de la pila.
CONSTANT	Se emplea para definir constantes.	Palabra de definición.

Tabla de órdenes FORTH		
PALABRA	DESCRIPCION	TIPO
VARIABLE	Define una variable.	Palabra de definición.
@	Busca el contenido de un elemento en memoria.	Palabra de definición.
!	Sitúa un elemento en memoria.	Palabra de definición.
REDEFINE	Cambia el valor de una constante	Palabra de definición.
.	Imprime en la pantalla el valor del elemento situado en la cima de la pila.	Trabajo con la pantalla.
."	Imprime en pantalla un mensaje.	Trabajo con la pantalla.
EMIT	Presenta en pantalla un carácter dado por su correspondiente carácter ASCII.	Trabajo con la pantalla.
CLS	Borra la pantalla.	Trabajo con la pantalla.
AT	Sitúa el cursor en un punto dado de la pantalla.	Trabajo con la pantalla.
HOME	Coloca el cursor en la esquina superior izquierda de la pantalla.	Trabajo con la pantalla.
SPACE	Escribe en la pantalla un espacio en blanco.	Trabajo con la pantalla.
SPACES	Escribe en la pantalla los espacios en blanco que indique el número situado en la cima de la pila.	Trabajo con la pantalla.
AND	Realiza la operación AND.	Operadores lógicos.
OR	Realiza la operación OR.	Operadores lógicos.
XOR	Realiza la operación OR-exclusiva.	Operadores lógicos.

tarla, el ordenador escribirá en la pantalla el carácter cuyo código ASCII se encuentra en la cima de la pila. Por ejemplo:

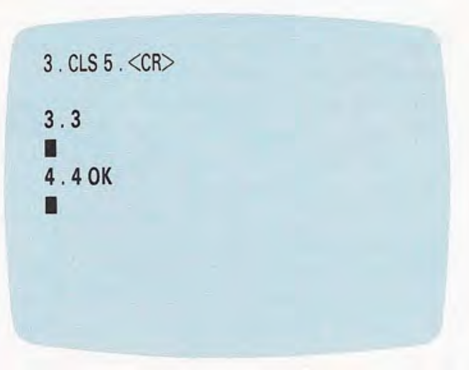
```
65 EMIT <CR>
65 EMIT A OK

48 EMIT <CR>
48 EMIT 0 OK
```

EMIT permitirá, pues, visualizar en pantalla los caracteres que conforman el repertorio del ordenador en uso. Dado que cada equipo tiene un juego de caracteres distinto, en mayor o menor grado, al de los demás, no es posible dar una tabla de códigos generalizable. No obstante, en cualquier caso, las cifras decimales tendrán el código comprendido entre los valores 48 y 57, y las letras desde el 65 en adelante (generalmente para las mayúsculas).

Hasta ahora se ha descrito cómo obtener una visualización en pantalla, aunque sin realizar ningún tratamiento específico de esta. El FORTH brinda algunas palabras especializadas en la operación con la pantalla; veamos a continuación las más importantes:

CLS Permite borrar la pantalla, eliminando de ella todo lo que anteriormente se encontraba escrito en la misma.



AT Sitúa el cursor de impresión en el punto especificado por los dos números más próximos a la cima de la pila. El más cercano a la cima, indica la columna, y el siguiente la fila en la que ha de realizarse la impresión. Por ejemplo:



Operadores lógicos

Tal y como establece la teoría de cálculo en el sistema binario, las operaciones lógicas se realizan bit a bit. El FORTH cuenta, habitualmente, con tres operadores lógicos fundamentales: AND, OR y XOR.

AND: Corresponde a la operación de producto lógico (AND).

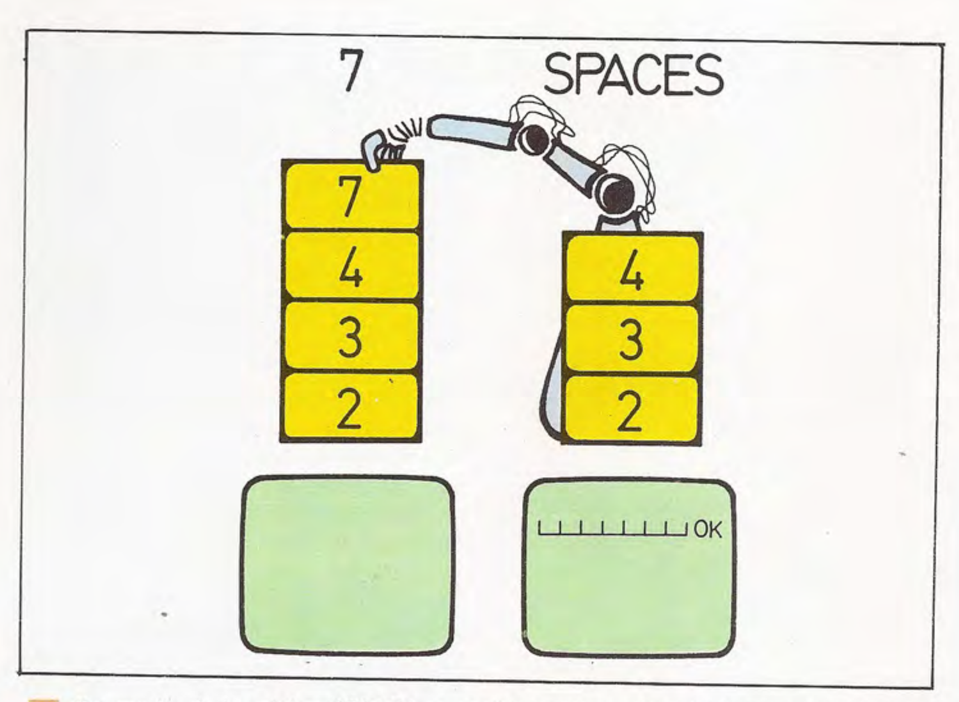
OR: Corresponde a la operación de suma lógica (OR).

XOR: Corresponde a la operación de suma lógica exclusiva (OR-exclusiva).

Veamos algún ejemplo al respecto:

```
2 7 AND . <CR>
2 7 AND .2 OK

2 7 OR . <CR>
2 7 OR .7 OK
```



Por medio de la palabra SPACES es posible ordenar a la máquina que trace una determinada zona de espacios en blanco sobre la pantalla.

2 7 XOR . <CR>
2 7 XOR .5 OK

HOME Traslada el cursor de impresión a la esquina superior izquierda de la pantalla.

SPACE Imprime en pantalla un espacio en blanco.

SPACES Visualiza una zona de espacios en blanco coincidente con el número situado en la cima de la pila:



Historia del FORTH

Aunque el lenguaje FORTH ya existía anteriormente en forma de rutinas sueltas, este no pasó a convertirse en un auténtico lenguaje de programación normalizado hasta el año 1969. Fue Charles H. Moore, del National Radiostronomy Observatory, localizado en Charlottesville (Virginia, U.S.A.), quien lo desarrolló y propuso como herramienta de trabajo destinada a realizar programas para controlar los grandes radiotelescopios empleados en astronomía.

En aquellos tiempos, el dominio correspondía a los grandes ordenadores. Y sobre estos ordenadores nació el FORTH.

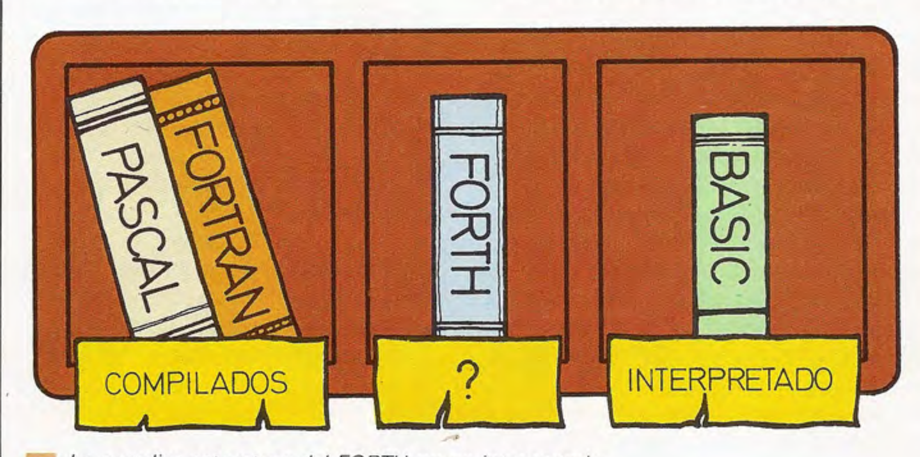
Más tarde, con la aparición de los microordenadores, el FORTH ha recibido un notable impulso al adaptarse fácilmente a ellos. De hecho, existen ordenadores domésticos que emplean el FORTH como lenguaje residente; tal es el caso del JUPITER ACE.

Este lenguaje está orientado hacia el desarrollo rápido de programas. Sus primeras aplicaciones se concretaron en el campo de la astrofísica. Campo en el que la aportación de este lenguaje ha sido muy grande, llegando a ser adoptado como lenguaje oficial por la Unión Astronómica Internacional.

En la actualidad, otro terreno que parece abonado para el FORTH es el control de procesos industriales, y dentro de él, el mundo de la robótica. Sus facultades en este ámbito derivan de la facilidad para crear rápidamente un programa a partir de una serie de rutinas independientes.

La filosofía del FORTH se distancia de la propia de otros lenguajes de uso más común; razón ésta por la que ha sido considerado siempre como un lenguaje difícil. Lo cierto es que se trata de un lenguaje ciertamente peculiar; sin embargo, no resulta mucho más difícil de aprender que cualquier otro si se parte de cero. En todo caso, sus características de modularidad, estructuración e interactividad, hacen del FORTH un lenguaje de indudable interés y atractivo.

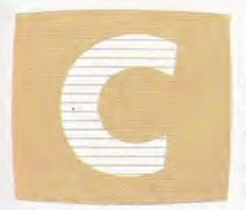
Pasado algún tiempo desde su nacimiento, y en un intento de unificar criterios con respecto a este lenguaje, apareció el F.I.G. Forth (Forth Interest Group). A esta siguieron otras versiones, más recientes, cuyo número da idea del creciente interés que existe por este lenguaje. En la actualidad existe una firma americana, FORTH INC, dedicada a la comercialización de todo lo relativo a este interesante lenguaje. Cabe afirmar que hoy en día pueden encontrarse versiones de FORTH para prácticamente todos los microordenadores dél mercado.



La peculiar estructura del FORTH no es interpretada ni compilada... ¡sino todo lo contrario!

FORTH (2)

Palabras y estructuras de control



omo ya se explicó en el primer capítulo de esta serie, el FORTH es un lenguaje que no trabaja ni en forma compilada ni interpretada. Realmente, opera de acuerdo a dos modalidades cuya intervención depende de la tarea en curso; éstos son ejecución inmediata y compilación.

Hasta el momento, todos los ejemplos vistos correspondían a la modalidad de trabajo en ejecución inmediata. Veamos ahora como también es posible operar en modo compilación.

En modo de ejecución inmediata, cada palabra introducida en la máquina es buscada en el diccionario y ejecutada inmediatamente. El modo compilación se emplea para proceder a la definición de nuevas palabras FORTH. En efecto, ahora la mayor parte de las palabras que se introducen no son ejecutadas; sino que, a través de la definición de la nueva palabra, se introduce en el diccionario una referencia a las palabras elementales que la componen. Tras la compilación, la nueva palabra pasa a formar parte del diccionario y puede ser invocada en cualquier momento; ya sea de forma inmediata, o bien por efecto de una referencia a la misma en otra definición (esto es, en modo de compilación).



El diccionario FORTH está compuesto por el conjunto de palabras utilizables en las tareas de programación y sus correspondientes definiciones.

Como definir nuevas palabras

La definición de palabras es el método que permite realizar la programación, propiamente dicha, en lenguaje FORTH. Gracias a la estructura del diccionario que hace posible emplear palabras previamente definidas para construir las nuevas palabras, el diccionario se ve ampliado, tanto en extensión como en potencia, a medida que se va trabajando con este lenguaje.

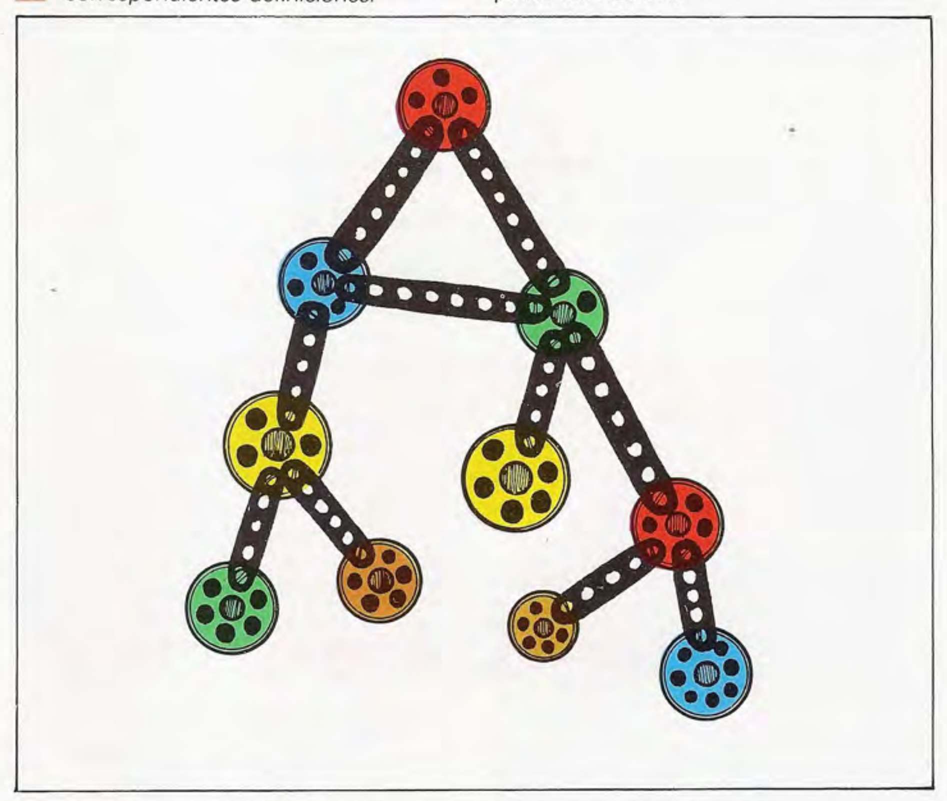
Para definir una nueva palabra es necesario proporcionar al ordenador una definición exacta de su significado. Para que éste sepa que el programador se dispone a darle la definición de una palabra, es preciso advertirlo mediante una orden representada por la palabra «:».

Una vez que el ordenador sabe que se encuentra en disposición de definir una palabra, el siguiente paso será comunicarle de qué palabra se trata; es decir: cuál es el nombre de la nueva palabra. Dicho nombre se escribirá a continuación.

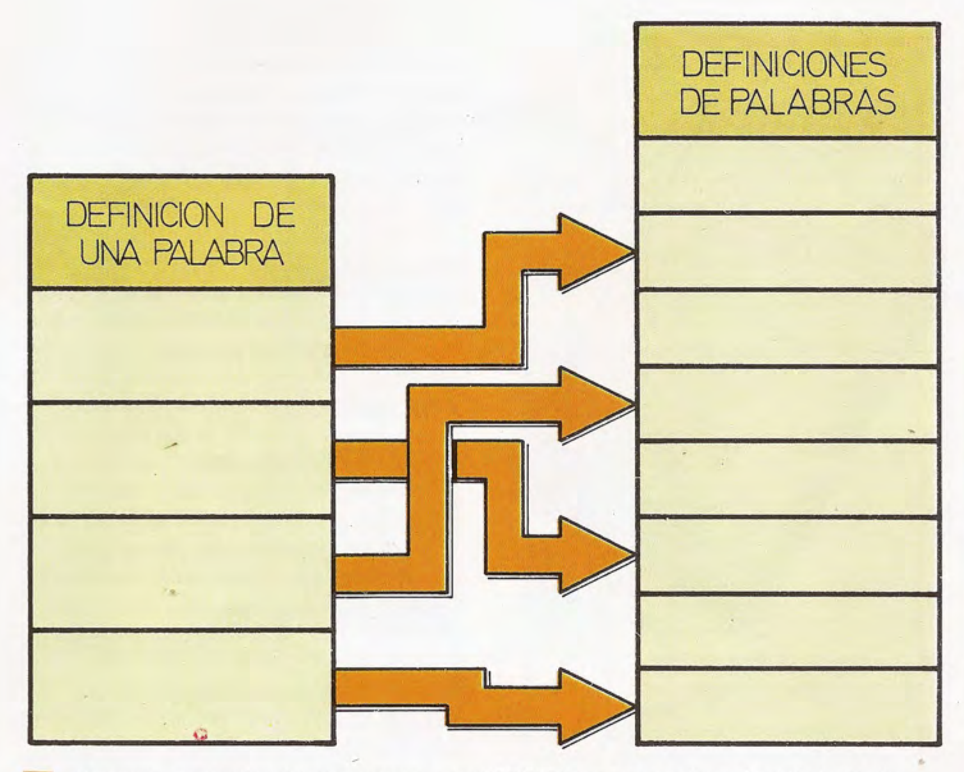
Es muy importante saber que el referido nombre puede contener cualquier símbolo que forme parte del repertorio de caracteres del ordenador, exceptuando a los símbolos gráficos y espacios en blanco. La longitud del nombre varía de una a otra versión del lenguaje FORTH, si bien, por lo general, suele situarse en torno a los 30 caracteres.

Una vez aportado el nombre, ha de procederse ya a la definición de la nueva palabra propiamente dicha; para lo cual el programador puede basarse tanto en las palabras que aporta el traductor FORTH, como en las que él hubiera definido previamente. El final de la definición de la nueva palabra ha de indicarse al ordenador mediante la palabra delimitadora «;».

Veamos un ejemplo de definición de palabras FORTH.



En el lenguaje FORTH las palabras se van definiendo de una forma jerárquica: la definición de una palabra puede edificarse en base a definiciones de palabras ya existentes.



El modo «compilación» se emplea en el FORTH para proceder a la definición de nuevas palabras. Por efecto de la definición de una palabra, se introduce en el diccionario FORTH una referencia a las palabras elementales que la componen.

En primer lugar, se trata de definir una palabra que permita sumar dos unidades al número situado en la cima de la pila:

```
:PALABRA<CR>
2+<CR>
;<CR>
```

Tras accionar por tercera vez la tecla RETURN (CR, o retorno de carro), el ordenador responderá con el siguiente mensaje por pantalla:

```
:PALABRA 2+;OK
■
```

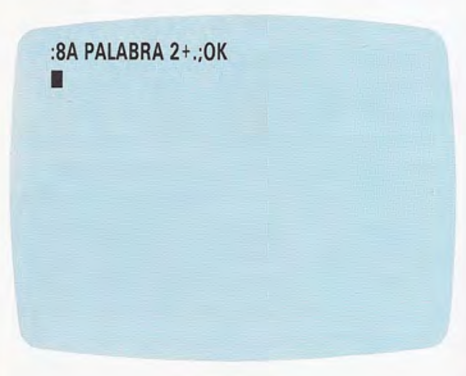
La respuesta será:

```
3 PALABRA.5 OK
```

Para trabajar con una palabra en modo compilación es necesario definir una nueva palabra que la contenga. En nuestro caso añadiremos otra palabra al diccionario FORTH. Esta nueva palabra realizará la misma función que la anterior, aunque incluiremos en su definición la orden de salida por pantalla del resultado. El nombre otorgado a la nueva palabra a definir será, por ejemplo, el de 8A. Como se observa, el nombre comienza con un número. Este es un hecho un tanto extraño para las personas acostumbradas a otros lenguajes de programación; si bien, en FORTH ello resulta de lo más habitual de hecho. El nombre de una palabra FORTH puede constar, si así se desea, únicamente de números o incluso de símbolos de puntuación.

```
:8A<CR>
PALABRA<CR>
2+.<CR>
;<CR>
```

Tras introducir la nueva palabra definida, la pantalla mostrará el siguiente mensaje:





Al modificar la definición de una palabra o proceder a la definición de la misma, ésta se almacena por encima de la antigua. Las nuevas referencias a dicha palabra afectarán a la definición más reciente de la misma. Sin embargo, las antiguas palabras seguirán existiendo. Para actualizar estas definiciones y eliminar las antiguas, es necesario emplear la palabra REDEFINE.

Con ello, el ordenador informa que la nueva palabra ha pasado a formar parte

del diccionario, y puede ya utilizarse libremente:

```
3 8A<CR>
3 8A 7 OK
█
```

Con lo cual, la nueva palabra queda ya incluida en el diccionario y puede ser utilizada a voluntad.

Para escribir la definición de una palabra no sólo puede emplearse el método aplicado en el ejemplo anterior. Existe otra posibilidad alternativa que consiste en escribir la definición a modo de una serie de palabras, una tras otra, separadas por espacios. En el caso del ejemplo descrito, la definición también podría haberse realizado como sigue:

```
:PALABRA 2+;<CR>
```

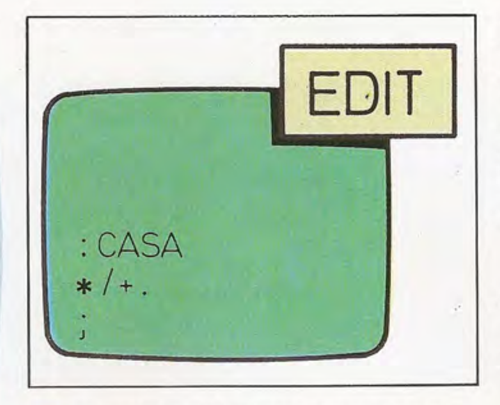
Ambos métodos de definición no muestran, en principio, ninguna diferencia aparente; en pantalla obtendremos el mismo mensaje que en el caso anterior.

Veamos a continuación cómo puede utilizarse la palabra definida. De trabajar en modo inmediato, es suficiente con introducir el nombre de la palabra en cuestión para que ésta se ejecute. Para ello, en primer lugar, se depositará en la pila un valor, y tras ello, ordenaremos a la máquina que visualice en la pantalla el número situado en la cima de la pila:

```
3 PALABRA.<CR>
```

Por supuesto, las palabras definibles pueden ser mucho más complejas que la contemplada en el ejemplo. En sucesivos ejemplos se verán definiciones de palabras cada vez más evolucionadas, a medida que se avance en el conocimiento de los bucles, decisiones y otros tipos de estructuras.

Llegados ya a este punto, resulta interesante poder conocer las palabras



La palabra EDIT traslada la definición de una palabra al buffer de entrada, permitiendo con ello realizar sobre la misma las modificaciones que sean necesarias.

que existen en el diccionario que estamos utilizando. El propio lenguaje FORTH incluye una palabra al efecto capaz de mostrar por pantalla todas las palabras existentes en el diccionario. Se trata de la palabra VLIST. Una vez introducida esta palabra, y tras pulsar la tecla de retorno de carro, aparecerán en la pantalla todas las palabras que componen el diccionario.

También es conveniente saber cómo es posible incluir dentro de una palabra comentarios que más tarde servirán como orientación a la hora de efectuar posibles modificaciones. Sencillamente, se escribirá el comentario que se desee encerrado entre paréntesis. Por ejemplo:

```
:MENSA<CR>
(GENERACION DE UN MENSAJE)<CR>
."ESTE ES EL MENSAJE"<CR>
;<CR>
```

La respuesta en pantalla coincidirá con:

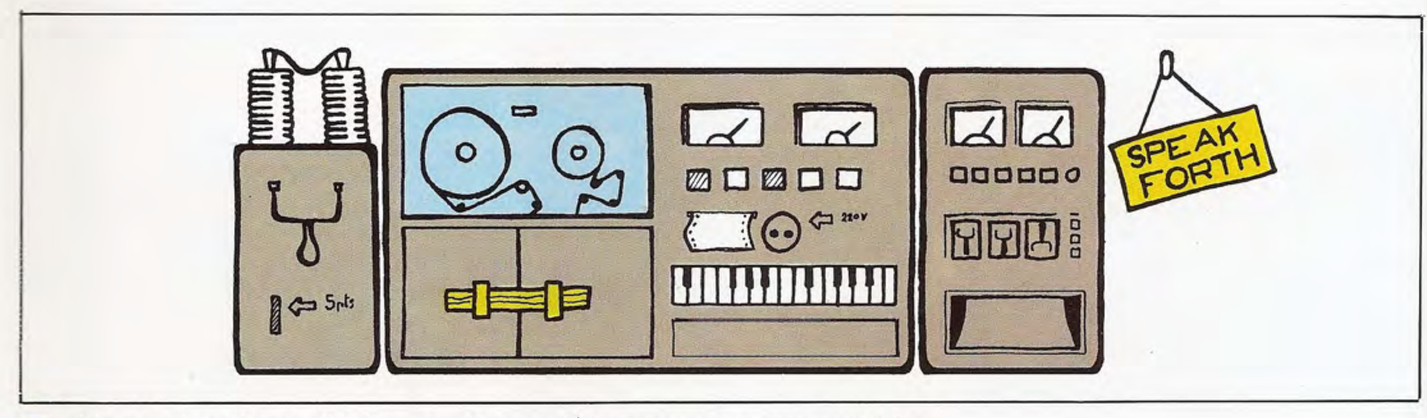
```
:MENSA (GENERACION DE UN MENSAJE)
 ."ESTE ES EL MENSAJE";OK
```

Desde luego, podemos ejecutar la nueva palabra definida de forma inmediata:

```
MENSA<CR>
ESTE ES EL MENSAJE OK
█
```



FORGET es la palabra que se utiliza para eliminar selectivamente palabras del diccionario.



Aún cuando en la actualidad invade el terreno de los microordenadores, el FORTH es un lenguaje que se creó sobre grandes equipos.

Cambiando definiciones de palabras

La definición de una nueva palabra no es algo inamovible, sino que una palabra puede ser borrada del diccionario o cambiada cuando así se desee; aunque, por supuesto, hay que tener en cuenta el hecho de que pueden existir otras palabras que hagan referencia a la palabra sujeta a modificación.

En una determinada situación, puede resultar interesante conocer cuál es la definición de una cualquiera de las palabras que existen en el diccionario. Para ello basta con ejecutar la palabra LIST, cuyo formato coincide con el siguiente:

```
LIST<palabra><CR>
```

Un vocabulario surtido para la creación de bucles facilitará la puesta a punto de estructuras fundamentales en el ámbito de la programación.

Donde <palabra> es la palabra cuya definición se desea visualizar en la pantalla. Por ejemplo:

```
LIST PALABRA<CR>

:PALABRA
2+
;
 OK
```

También puede resultar de interés borrar algunas de las palabras que forman parte del diccionario. Para ello también existe una palabra FORTH especializada: se trata de FORGET, cuya formulación debe ofrecer el siguiente aspecto:

```
FORGET<palabra><CR>
```

En donde <palabra> es la palabra que deseamos que desaparezca del diccionario. Evidentemente no es posible hacer desaparecer ninguna de las palabras incluidas originalmente en el propio traductor de FORTH.

Veamos un ejemplo:

```
FORGET MENSAJE<CR>
```

La respuesta del ordenador será:

```
FORGET MENSAJE OK
```

Si a continuación se intenta visualizar la definición de la palabra borrada, la respuesta de la máquina coincidirá con un mensaje de error (LIST ERROR).

También es posible introducir modificaciones en la definición de una pala-

bra ya existente. Al respecto, basta con llevar la definición de dicha palabra al buffer de entrada (hay ordenadores en los que es suficiente con que la definición se encuentre en la pantalla). Para conseguirlo, hay que hacer uso de la palabra EDIT, cuyo formato es el siguiente:

```
EDIT<palabra><CR>
```

Donde <palabra> es la palabra que se desea trasladar al buffer de entrada en orden a realizar en ella las modificaciones pertinentes.

Veamos a continuación un ejemplo ilustrativo de su puesta en práctica. Se empezará definiendo una palabra que eleve un número al cuadrado.

```
:CUADRADO<CR>
*.<CR>
;<CR>
```

En esta definición hay un error, ya que lo lógico sería introducir el número del que se desea obtener el cuadrado, y que el ordenador entregara la respuesta; algo semejante a:

```
3 CUADRADO<CR>
```

Sin embargo, y lamentablemente, el ordenador entrega por toda respuesta un mensaje de error. Ello se debe a que se ha ordenado a la máquina que multiplique dos números; estos serán los situados más cerca de la cima de la pila. Como quiera que la definición tan sólo deposita uno de ellos en la pila, el ordenador no es capaz de encontrar el segundo número. Para evitar tal situación, es preciso incluir en la definición una palabra que duplique la cima de la pila (la instrucción DUP, por ejemplo).

Para proceder a la modificación de la palabra CUADRADO, se empezará escribiendo la instrucción:

```
EDIT CUADRADO<CR>
```

Acto seguido, es posible ya acondicionar la definición de la palabra hasta dejarla de la siguiente forma:

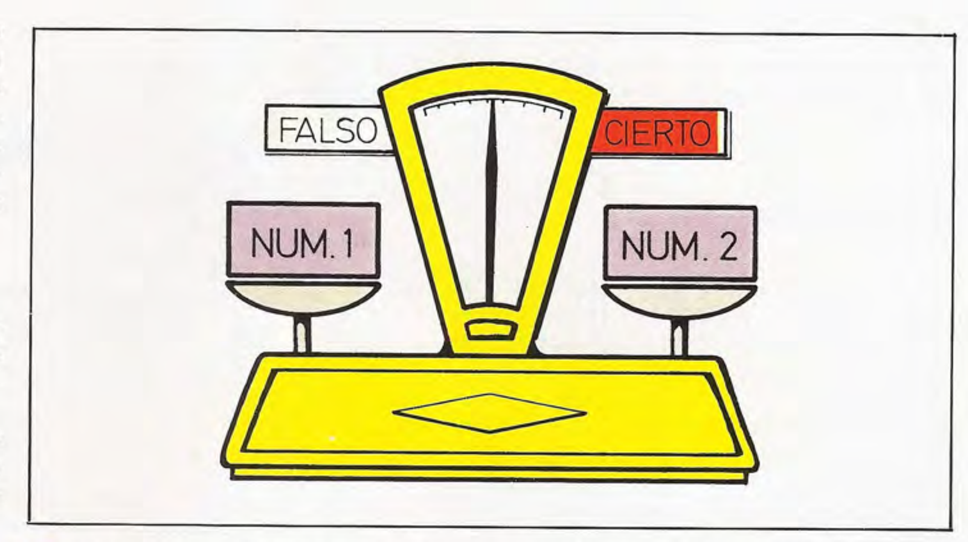
```
:CUADRADO
DUP*.
```

No obstante, si ahora se hace uso de la palabra VLIST, el programador descubrirá que existen dos definiciones de la misma palabra.

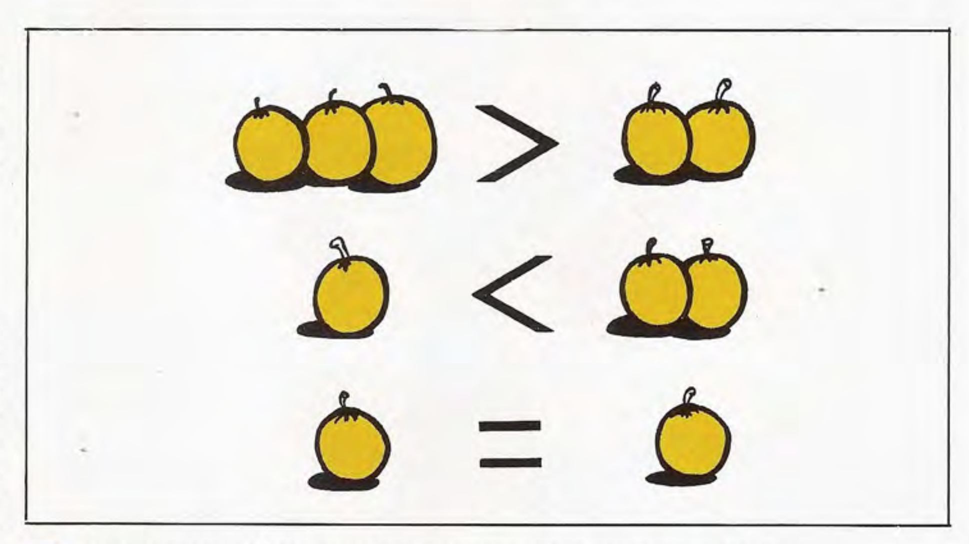
En principio, y en el caso de operar en el modo inmediato, tal situación no plantea problema alguno. No obstante, si antes de realizar dicha modificación se hubiera definido otra palabra —por ejemplo TARA— que hiciera uso de la palabra CUADRADO, tras efectuar la modificación se observaría que la palabra TARA sigue utilizando la primitiva definición de la palabra CUADRADO. Para actualizar esta referencia y, en consecuencia, eliminar las definiciones redundantes, se hace necesario el empleo de la palabra REDEFINE. Esta actualiza las versiones de una palabra y las referencias que otras palabras hagan a la misma. Su formato es el siguiente:

```
REDEFINE<palabra><CR>
```

A partir de los conocimientos expues-



Los operadores lógicos permiten comparar dos elementos, obteniendo un resultado que puede adoptar tan sólo dos valores: verdadero o falso.



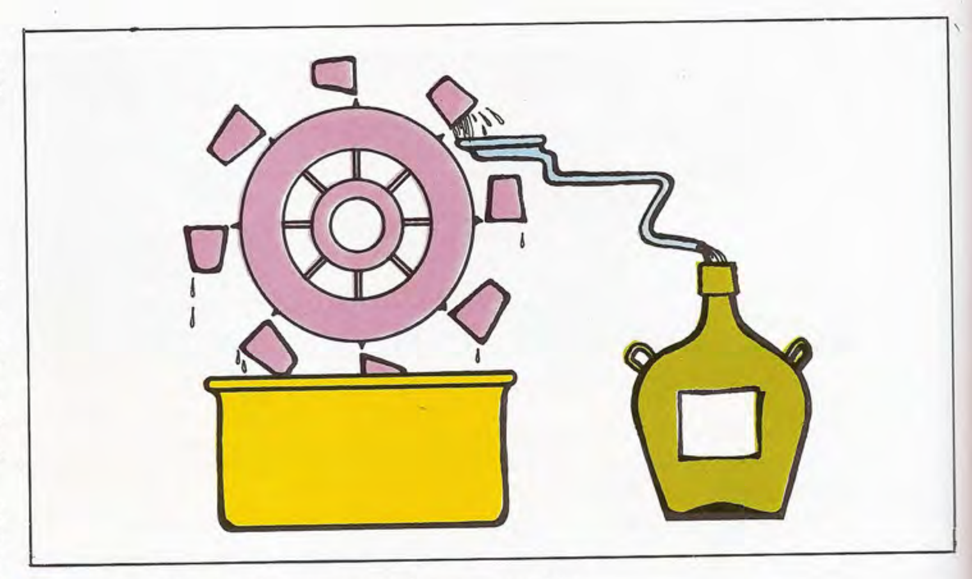
Los operadores lógicos resultan de gran utilidad para la comparación de pares de elementos. Su resultado puede utilizarse como condición a evaluar para tomar una decisión.

tos en los párrafos anteriores, es posible ya ir pensando en la elaboración de los primeros programas en lenguaje FORTH.

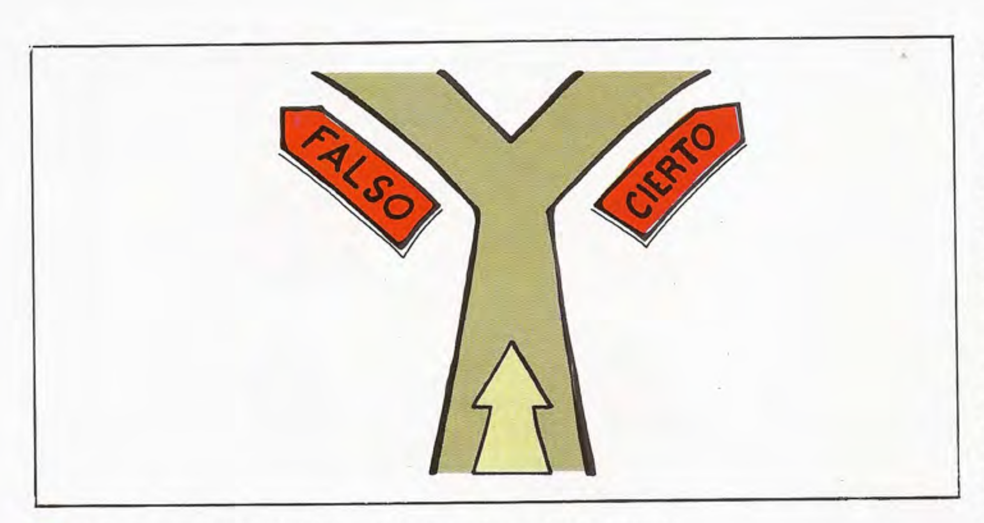
Realizando comparaciones

En esencia, una decisión consiste sencillamente en la elección de una entre dos o más posibles alternativas. Cada una de las opciones a seleccionar tiene asociado un proceso diferente, ya sea en su totalidad o en parte, respecto al asociado a las restantes alternativas que se ofrecen.

Toda decisión se fundamenta en una determinada condición a evaluar. En el caso de disponer de dos alternativas, si el resultado de evaluar la condición impuesta es *cierto,* se actuará en un sentido, mientras que si es *falso* se actuará en otro. A raíz de tal razonamiento, parece obvio que lo primero que es preciso conocer es cuáles son las condiciones que pueden darse al ordenador para que éste proceda a su examen y, en consecuencia, tome una decisión. En el caso del lenguaje FORTH, las condiciones admisibles son las establecidas por medio de los operadores cuyas palabras FORTH se definen a continuación:



Los bucles permiten repetir de forma automática secuencias de operaciones predefinidas.



Las instrucciones de bifurcación permiten seleccionar uno de entre los dos caminos posibles; la elección dependerá del cumplimiento o no de la condición que se imponga.

= Toma los dos números situados más cerca de la pila y examina si son iguales.
< Toma los dos números de la cima de la pila y comprueba si el segundo es menor que el primero. Ello equivale a examinar si el número situado en la cima de la pila es mayor que el que se encuentra a continuación de éste.
> Toma los dos números de la cima de la pila y comprueba si el segundo número es mayor que el primero.
0= Toma el número situado en la cima de la pila y comprueba si es cero.
0< Comprueba si el número situado en la cima de la pila y comprueba si es menor que cero.
0> Toma el número situado en la cima de la pila y comprueba si es mayor que cero.

El resultado de cualquiera de estas evaluaciones puede tomar dos valores: VERDADERO o FALSO. En el caso de ser verdadero, el ordenador deposita en la cima de la pila un uno, mientras que si es falso el valor depositado será un cero.

En este punto resulta conveniente señalar la posibilidad que existe de combinar dos o más condiciones para obtener un único resultado. Veamos algún ejemplo, expresado literalmente, ilustrativo de condiciones evaluables de acuerdo a la referida posibilidad:

«¿El número A es mayor que B y a su vez el número D es menor que B».
«¿El número A es mayor que B o A es igual a B».

El medio para establecer estas condiciones compuestas, derivadas de la síntesis de dos o más condiciones elementales, lo aportan los operadores lógicos. Como ejemplo veamos cómo se escribirían las condiciones enunciadas anteriormente en el caso de sustituir los datos A, B y D por ciertos valores numéricos constantes y determinados.

```
4 7>3 7<AND.<CR>
```

```
4 7>3 7<AND.0 OK

5 5>5 5=OR.<CR>

5 5>5 5=OR.1<CR>
```

Elección del camino adecuado

Una vez que se conoce la forma de imponer una condición, es preciso saber cómo aplicar su examen a la decisión entre dos alternativas.

Existen tres palabras claves del diccionario FORTH ligadas con las estructuras de decisión. Estas son: IF, ELSE y THEN. La primera de ellas se emplea para ordenar a la máquina que tome una decisión. La segunda delimita el primer bloque de instrucciones, justamente el que se ejecutará en el caso de que la condición sea cierta; dicha sección de instrucciones será, por lo tanto, la comprendida entre las palabras IF y ELSE.

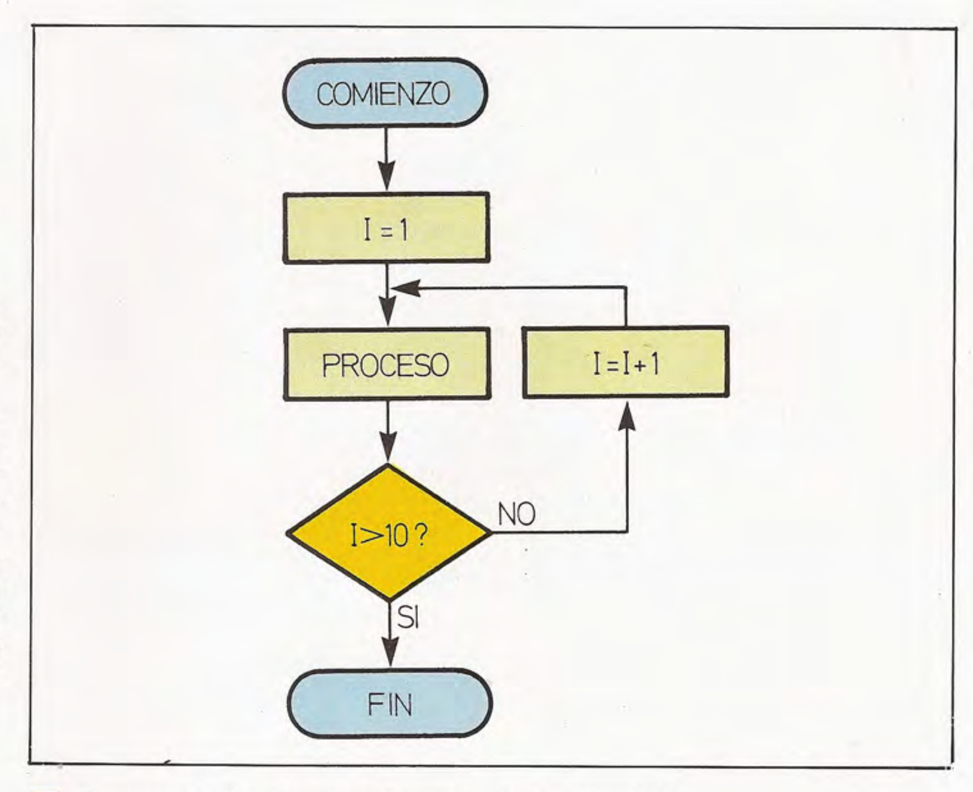
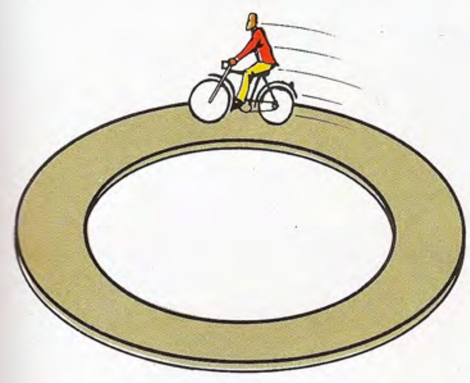


Diagrama de flujo correspondiente a un bucle controlado.



La ejecución de un bucle DO/LOOP puede compararse con una carrera de velocidad que termina una vez que se han completado un determinado número de vueltas al circuito. A su vez, los bucles del tipo DO/UNTIL son equiparables a las «24 horas de Le Mans»: hay que correr durante 24 horas sin importar el número de vueltas que se den al circuito.

El segundo bloque de instrucciones, el que se ejecutará en el caso de que la condición no se cumpla, queda delimitado entre las palabras ELSE y THEN.

Sin lugar a dudas, un ejemplo vale más que mil palabras; veamos pues un ejemplo de aplicación de una de estas condiciones. Para ello vamos a definir una palabra que llamaremos EXAMEN. En caso de que la cima de la pila contenga un número inferior a 50, el ordenador exhibirá en la pantalla el mensaje SUSPENSO; mientras que si dicho número es mayor que 50 aparecerá el mensaje APROBADO.

```
:EXAMEN<CR>
50<<CR>
  IF<CR>
      ."SUSPENSO"<CR>
  ELSE<CR>
      ."APROBADO"<CR>
  THEN<CR>
;<CR>
```

A modo de respuesta, el ordenador mostrará en pantalla la siguiente confirmación.

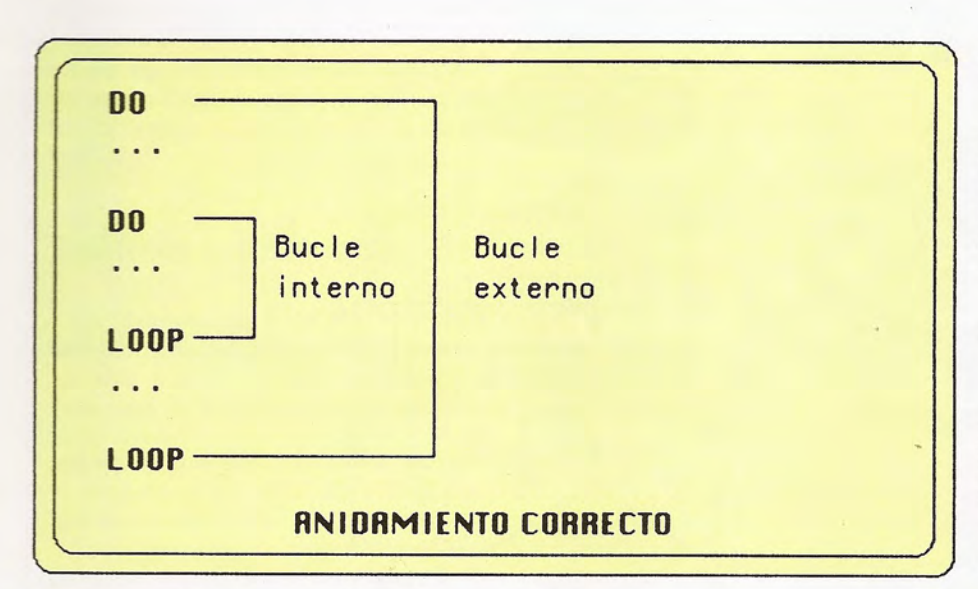
```
:EXAMEN 50<IF."SUSPENSO"ELSE."
APROBADO"THEN ; OK
```

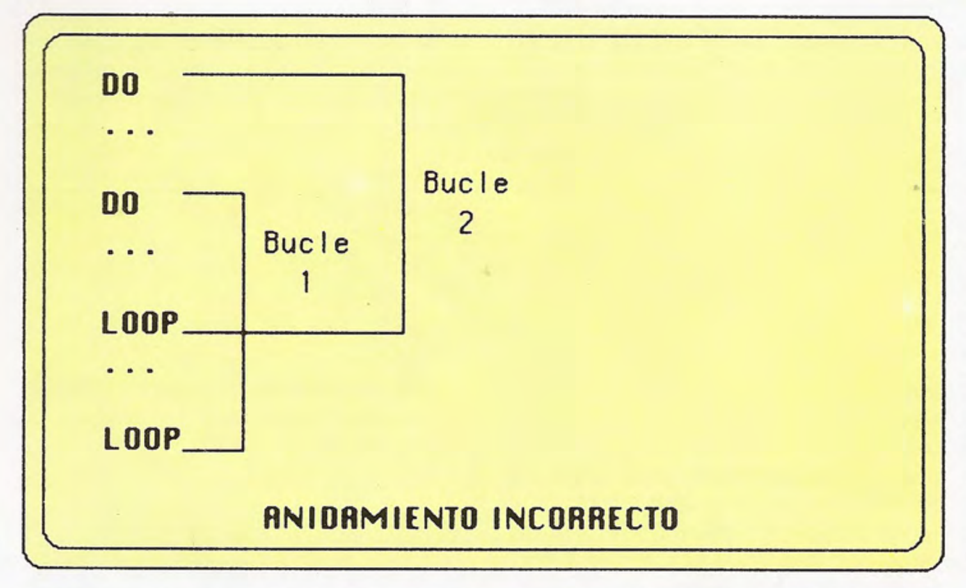
Una vez definida la palabra EXAMEN, es posible ya utilizarla en la práctica. Por ejemplo:

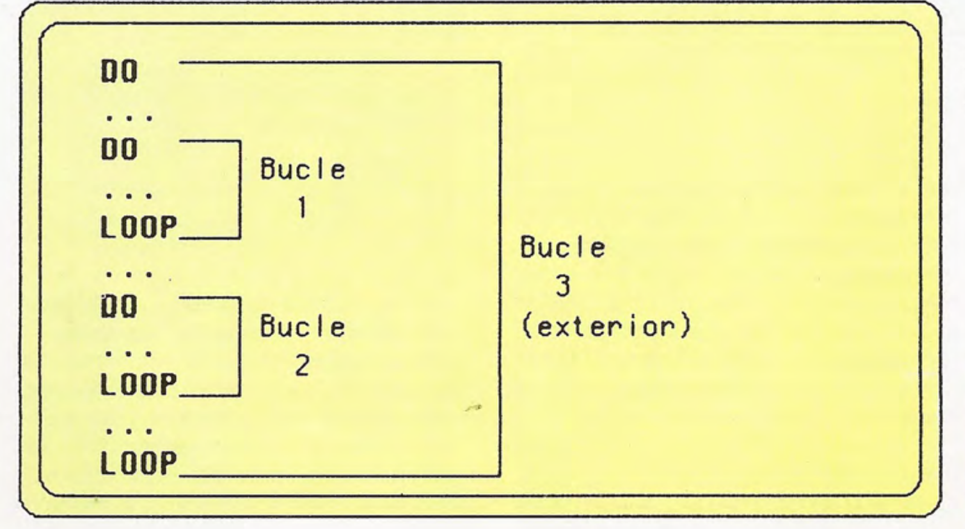
```
40 EXAMEN<CR>
40 EXAMEN SUSPENSO OK

70 EXAMEN<CR>
70 EXAMEN APROBADO OK
```

En ocasiones es posible que la evaluación de la condición impuesta no conduzca a dos procesos distintos entre los que elegir; en efecto, uno de los procesos puede ser nulo y dar lugar a que prosiga, sencillamente, la ejecución del programa. En semejante situación no es necesario incluir la palabra ELSE utili-







zada en el ejemplo al definir la palabra EXAMEN. Ejemplo:

```
: ELIGE<CR>
DUP 13=<CR>
IF<CR>
DROP 12<CR>
THEN<CR>
;<CR>

:ELIGE DUP 13=IF DROP 12 THEN ; OK
```

La nueva palabra definida, ELIGE, examinará el número situado en la cima de la pila, y en caso de que éste sea igual a 13 lo sustituirá por el valor 12. Este es un ejemplo elemental de estructura de decisión en la que está ausente la zona ELSE.

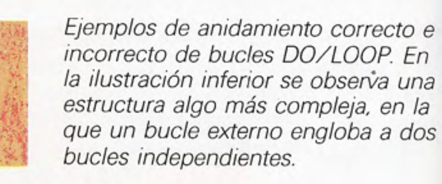
Bucles elementales

Un bucle consiste en la repetición cíclica y bajo control de una secuencia de instrucciones. Una forma elemental de construir un bucle es la que ilustra el siguiente ejemplo:

```
:BUCLE<CR>
DUP 10<<CR>
IF<CR>
1+DUP.BUCLE<CR>
ELSE<CR>
."FIN"<CR>
;<CR>

:BUCLE DUP 10 IF 1+DUP.BUCLE ELSE."
FIN" ; OK
```

La palabra definida recoge el número situado en la cima de la pila y comprueba si es menor que 10; de ser cierto, incrementa su valor en una unidad, e in-



Ejemplos de anidamiento correcto e incorrecto de bucles DO/LOOP. En la ilustración inferior se observa una estructura algo más compleja, en la que un bucle externo engloba a dos bucles independientes.

voca de nuevo a la propia palabra BUCLE. El proceso se detendrá cuando se alcance un valor superior a 10.

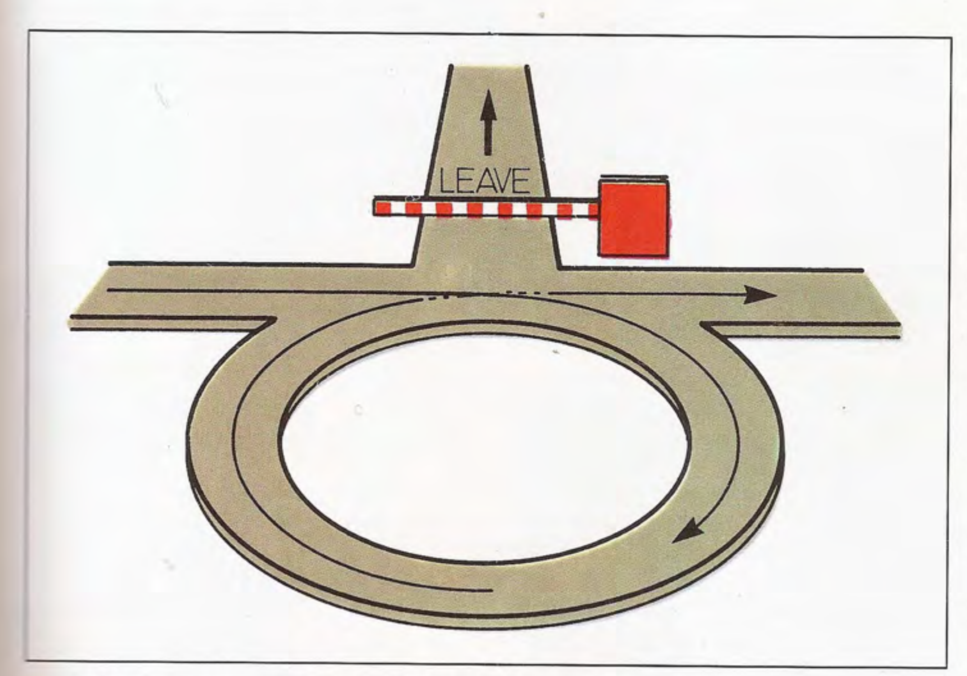
```
2 BUCLE CR
2 BUCLE 3 4 5 6 7 8 9 10 FIN OK
```



EXIT permite abandonar la ejecución de un bucle «por la puerta falsa».

Bucles controlados

No es imprescindible construir los bucles «a mano», sino que el programador tiene a su alcance algunas estructuras,



LEAVE es una palabra del vocabulario FORTH cuya llamada permite abandonar la ejecución de un bucle.

incluidas en el vocabulario FORTH, que permiten el control de bucles. Tal es el caso de la estructura DO/LOOP, cuya especialidad es hacer que se repita la ejecución de una secuencia de instrucciones un determinado número de veces. Para ello, es necesario prefijar los valores inicial y final del contador de bucles. Ambos números deben depositarse previamente en la pila. El primero a situar en la pila debe ser el correspondiente al valor final y el segundo el valor inicial.

La secuencia de instrucciones a repetir (instrucciones del bucle) debe estar delimitada entre las palabras clave DO y LOOP.

Veamos a continuación un ejemplo; se trata del mismo bucle construido anteriormente, si bien, ahora intervendrán en el mismo las palabras DO/LOOP.

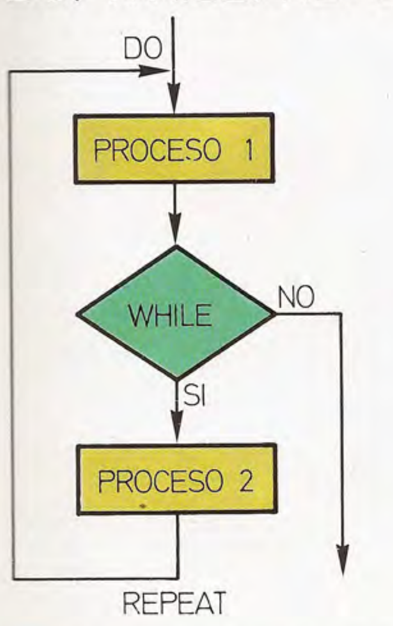
```
:BUCLE<CR>
2 10 2<CR>
DO<CR>
1+DUP.<CR>
LOOP<CR>
;<CR>

BUCLE 2 10 2 1+DUP.LOOP ; OK
```

Y aquí está el resultado de invocar a la palabra BUCLE definida:

```
BUCLE<CR>
BUCLE 3 4 5 6 7 8 9 10 OK
```

La técnica más sencilla para la creación de un bucle se concreta en partir de un valor de referencia prefijado, valor que debe incrementarse en una unidad cada vez que sea recorrido el bloque de instrucciones que dan cuerpo al bucle; la secuencia de ejecución cíclica terminará, saliendo del bucle, cuando el valor de referencia —contador de ciclos— alcance el valor final, también



Las palabras DO/WHILE/REPEAT actúan sobre dos procesos que se ejecutarán cíclicamente hasta que deje de cumplirse una determinada condición. Tal como muestra el diagrama de flujo, el proceso 2 no se ejecutará al salir de la estructura.

prefijado. En ciertas ocasiones, es necesario crear un bucle en el que, en cada pasada, el incremento del valor de referencia no sea unitario, sino de varias unidades; e incluso puede interesar que el valor del incremento sea positivo o negativo.

Para contemplar esta posibilidad, el lenguaje FORTH dispone de una palabra adecuada para sustituir a la palabra LOOP que encontrábamos anteriormente. Esta nueva palabra (+LOOP) incrementará el valor del índice en tantas unidades como indique el número que se encuentre en la cima de la pila.

Veamos un sencillo ejemplo:

```
:CANGREJO<CR>
—1 10<CR>
DO<CR>
—1<CR>
+LOOP<CR>
;<CR>
```

La ejecución de la palabra que acabamos de crear, CANGREJO, no produce ningún resultado en la pantalla. Para observar algún efecto en la visualización vamos a introducir una nueva palabra FORTH. Se trata de la palabra I, cuya función consiste en copiar el valor del índice, que se encuentra en la pila de retorno, a la pila de datos, permitiendo de este modo realizar alguna operación con el referido índice.

En este caso, el índice utilizará para examinar la forma en la que evoluciona el recorrido del bucle. Al efecto, será preciso editar la palabra CANGREJO, previamente definida, y modificarla hasta que adopte la siguiente forma:

```
:CANGREJO
—1 10
DO
I.—1
+LOOP
;
```

Ahora, al invocar la palabra CANGREJO, la respuesta en pantalla será la siguiente:

```
CANGREJO<CR>
CANGREJO 10 9 8 7 6 5 4 3 2 1 0 OK
```

En efecto, se observa que tal y como se ha previsto, el índice evoluciona en sucesivos saltos de —1 unidad por cada pasada que se da al bucle.

Además de la palabra I, cuya misión es pasar un valor de la pila de retorno a la de datos, existen más palabras FORTH que permiten operar pasos de una a otra pila. Por ejemplo: la palabra >R, la cual permite pasar valores de la pila de datos a la de retorno, y la palabra R> que los transfiere de la pila de retorno a la pila de datos.

Bucles anidados

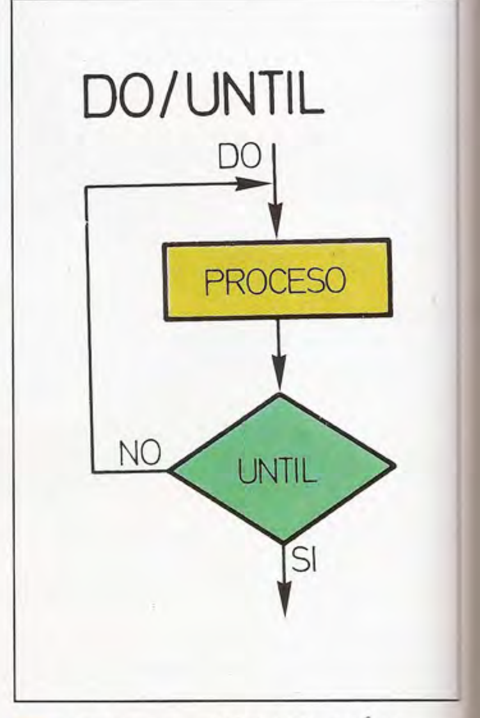
Al igual que la mayor parte de los lenguajes de alto nivel, también en el FORTH es posible construir bucles anidados. El anidamiento de bucles consiste, sencillamente, en introducir nuevos bucles dentro de otros más generales. La única condición que se impone al anidamiento de bucles es, precisamente, que tal anidamiento sea correcto. Es decir, si un bucle está incluido dentro de otro más amplio, es necesario que el bucle anidado comience y termine en el interior del bucle externo.

En el cuadro adjunto puede observarse algún ejemplo gráfico de anidamiento correcto e incorrecto.

En la figura adjunta, se observa que también es posible realizar anidamientos más complejos. El ilustrado corresponde al anidamiento de tres bucles en dos niveles: el bucle externo incluye a dos bucles elementales y encadenados en sucesión.

Veamos un ejemplo programado de anidamiento de bucles:

```
:BUCLE<CR>
5 1 <CR>
DO<CR>
I.5 1<CR>
```



La estructura DO/UNTIL permite repetir un proceso tantas veces como sea necesario que llegue a cumplirse la condición impuesta.

```
DO<CR>
I.<CR>
LOOP<CR>
CR<CR>
LOOP<CR>
;<CR>
```

En el ejemplo aparece una nueva palabra FORTH: CR, la cual ordena un retorno de carro en la pantalla; esto es: al ejecutar la palabra CR cambia la posición del cursor, pasando del punto en el que la dejó la anterior operación realizada en la pantalla al comienzo de la siguiente línea.

Al invocar a la palabra BUCLE:

```
BUCLE<CR>
```

se observará en la pantalla conectada al ordenador el resultado que sigue:

```
BUCLE 1 1 2 3 4 5
2 1 2 3 4 5
3 1 2 3 4 5
4 1 2 3 4 5
5 1 2 3 4 5 OK
```

Veamos un nuevo ejemplo:

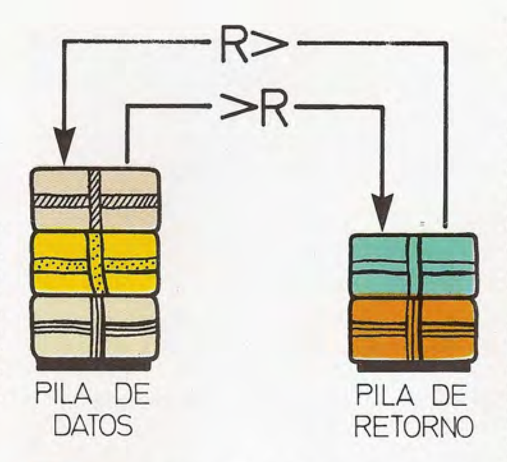
```
:FILAS<CR>
CR 1+1<CR>
DO<CR>
I0<CR>
DO<CR>
```



Comparaciones

Los operadores de comparación trabajan con números depositados en la pila, dando como resultado los valores uno (verdadero) o cero (falso).

- = Comparador de igualdad
- < Menor que
- > Mayor que
- 0= Comprueba si el número es igual a cero
- 0< Comprueba si el número es negativo
- 0> Comprueba si el número es positivo



Las palabras >R y <R permiten transportar datos de la pila de datos a la de retorno y viceversa.

```
."*"<CR>
LOOP<CR>
CR<CR>
LOOP<CR>
CR<CR>
;<CR>
```

Para invocar a la nueva palabra definida (FILAS), es necesario dejar en la pila un valor que indique el número de filas de asteriscos que se desea dibujar en la pantalla.

```
4 FILAS<CR>

4 FILAS
*
**
***
****
OK
```

Para abandonar un bucle durante su ejecución, es necesario ejecutar una instrucción LEAVE, la cual permite abandonar el ciclo en curso. Para ello, llevará el valor del índice al límite y originará la salida del bucle tras completar la pasada que se esté efectuando en ese preciso momento.

Bucles indefinidos

Los bucles que se han utilizado hasta el momento eran bucles controlados. Ello significa que el bucle se ejecuta un número de veces previamente establecido, antes de que dé comienzo la ejecución del bucle (excepto en el caso de que se utilice la palabra LEAVE).

Existen otros dos tipos de bucles cuyo funcionamiento difiere del descrito hasta ahora. Se trata de los bucles creados con el apoyo de las palabras BEGIN/UNTIL y BEGIN/WHILE/REPEAT. En esencia, estos bucles se ejecutan tantas veces como sea necesario hasta que se vea satisfecha una condición. Veamos cada uno de ambos casos por separado.

- Bucles BEGIN/UNTIL

Se ejecutarán las instrucciones encerradas entre las palabras BEGIN y UNTIL las veces que sea necesario hasta lograr que se verifique la condición impuesta. La evaluación del referido condicionante ha de realizarse antes de llegar a la instrucción UNTIL, de tal forma que al llegar al nivel de UNTIL, se encuentre en la cima de la pila el valor «verdadero» o «falso». Veamos un ejemplo:

IF/THEN/ELSE

Esta estructura permite tomar decisiones entre dos alternativas, en base a una condición impuesta. El proceso que se ha de ejecutar en el caso de que la condición sea cierta se detalla entre las palabras IF y ELSE, mientras que el proceso a ejecutar en caso de que no se cumpla la condición se especificará entre las palabras ELSE y THEN.

DO/LOOP

Permite definir bucles que se ejecutarán repetidamente un determinado número de veces, a partir de un valor inicial y hasta un valor final; estos dos valores deben depositarse previamente en la pila. El proceso que constituye el bucle debe escribirse entre las palabras clave DO y LOOP.

Tabla de valores FORTH		
PALABRA	DESCRIPCION	TIPO
:	Indica el comienzo de la definición de una palabra.	Palabra de definición
;	Indica el fin de la definición de una palabra	Palabra de definición
LIST<palabra>	Lista la definición de una palabra	Tratamiento de palabras
FORGET<palabra>	Borra una palabra del diccionario.	Tratamiento de palabras.
EDIT<palabra>	Edita la definición de una palabra.	Tratamiento de palabras.
REDEFINE<palabra>	Actualiza el diccionario y las referencias a una palabra.	Tratamiento de palabras.
=	Comparador de igualdad	Operador de comparación.
<	Menor que.	Operador de comparación.
>	Mayor que	Operador de comparación.
0=	Comprueba si el número es igual a cero.	Operador de comparación.
0<	Examina si el número es negativo.	Operador de comparación.
0>	Examina si el número es positivo.	Operador de comparación.
IF/THEN/ELSE	Estructura alternativa que permite elegir entre dos procesos.	Estructura alternativa.
DO/LOOP	Estructura repetitiva que permite ejecutar un proceso un número dado de veces.	Estructura repetitiva.
DO/+LOOP	Creación de un bucle, con un incremento sucesivo del índice distinto a la unidad	Palabras de control
LEAVE	Permite abandonar un bucle antes de que éste sea completado	Palabras de control
DO/UNTIL	Ejecuta un proceso hasta que se cumple la condición impuesta	Palabras de control
DO/WHILE/REPEAT	Ejecuta un proceso hasta que deje de cumplirse la condición establecida	Palabras de control
EXIT	Permite abandonar un bucle DO/UNTIL o DO/WHILE sin necesidad de que se satisfaga la condición impuesta	Palabras de control
I	Toma el valor del índice de la pila de retorno y los pasa a la pila de datos	Transferencia de datos
CR	Genera un retorno de carro	Impresión en pantalla
R>	Transfiere valores de la pila de retorno a la de datos	Transferencia de datos
>R	Transfiere valores de la pila de datos a la de retorno	Transferencia de datos

```
:+—<CR>
BEGIN<CR>
1+DUP .DUP 8<CR>
><CR>
UNTIL<CR>
;<CR>
```

Al invocar la ejecución de la palabra definida se obtiene la siguiente respuesta en pantalla:

```
1+—2 3 4 5 6 7 8 9 OK
```

La instrucción BEGIN/WHILE/REPEAT presenta algunas diferencias esenciales respecto a la instrucción UNTIL. Para empezar, en este nuevo caso la ejecución se detendrá precisamente cuando la condición deje cumplirse. Además, hay dos secciones dentro del bucle: una de ellas se ejecuta antes de examinar la condición (la zona comprendida entre BEGIN y WHILE), mientras que la otra sección se ejecutará tan sólo si la condición se cumple. En caso de que no se verifique la condición impuesta, la secuencia saltará directamente a la instrucción que sigue a la palabra REPEAT.

El siguiente ejemplo ilustra alguno de estos conceptos:

```
:—<CR>
BEGIN<CR>
2+DUP 16 <<CR>
WHILE<CR>
DUP.<CR>
REPEAT<CR>
;<CR>
```

Veamos cuál será el resultado de ejecutar la nueva palabra:

```
1—<CR>
```

Y la respuesta que se obtiene en pantalla es la siguiente:

```
1—3 5 7 9 11 13 15 OK
```

De estos bucles también es posible salir por la «puerta falsa» empleando la instrucción EXIT; al igual que ocurría con los bucles controlados en el caso de ejecutar la instrucción LEAVE.

FORTH (3)

Entrada de datos y bases de numeración



La finalidad de un programa no es otra que instruir al ordenador para que éste realice un determinado tratamiento de los datos aportados. Estos datos pueden estar incluidos dentro del programa, con lo cual el proceso resulta sencillo. No obstante, la situación más normal es la de construir el programa sin contar con los datos específicos a procesar. Ello supone que será preciso introducirlos cada vez que se ejecute el programa, ya sea previamente o bien durante el proceso.

En el primero de los casos es suficiente con colocar los datos en la pila, y ordenar las operaciones oportunas para que el programa sea capaz de tratar esos valores.

En el segundo caso los datos han de introducirse durante el proceso, y para ello es necesario utilizar las palabras que el FORTH brinda a tal efecto. Estas palabras constituyen, precisamente, el objeto del presente capítulo.



El diálogo con la máquina en FORTH se establece normalmente recurriendo al teclado como vía de entrada. En tal caso, entra en actividad el «buffer de entrada»: una zona de memoria que actúa a modo de intermediario entre el usuario y el ordenador.

Trabajando con el buffer de entrada

El buffer de entrada es una zona de memoria que actúa a modo de intermediario entre el usuario y el ordenador. Lo que el programador teclea al editar un programa va escribiéndose en el buffer, y pasa a la memoria del ordenador cuando se acciona la tecla de retorno de carro. En consecuencia, parece obvio que una forma de pasar datos a un proceso consiste, sencillamente, en explotar la función propia del buffer de entrada.

Existen dos instrucciones que sirven para introducir datos en el ordenador a través del buffer, éstas coinciden con las palabras QUERY y RETYPE. Ambas detienen el proceso en curso y facilitan el acceso al buffer. La diferencia entre ambas es que la primera empieza borrando el contenido previo del buffer de entrada, y a continuación permite acceder al mismo. La segunda no borra ni altera el contenido previo, por lo que será posible editar el contenido del buffer, introduciendo las modificaciones oportunas. En todo caso, ninguna de ambas palabras afecta a la pila. Desde luego, el siguiente paso es tratar el buffer de entrada de la forma más adecuada. Para ello, el FORTH dispone de cuatro palabras que será posible emplear según las necesidades específicas. Estas son:

LINE, NUMBER, FIND y WORD

La primera de ellas (LINE), toma sencillamente el contenido del buffer de entrada y lo interpreta, produciendo el mismo resultado que si se hubieran tecleado esas mismas palabras en modo inmediato. El uso de LINE resulta muy flexible, aunque no deja de plantear ciertos problemas, ya que la palabra o palabras introducidas en el buffer pueden originar tras su interpretación y ejecución un resultado no previsto e indeseado. Por ejemplo, el resultado puede ser la obtención de dos números, en lugar de uno solo como en principio podía estar previsto.

Veamos un ejemplo de definición en el que intervienen las palabaras clave QUERY y LINE:

```
: ENTRADA
CR
."INTRODUCE LO QUE DESEES"
QUERY LINE
CR
."YA SE HA EJECUTADO"
;
```

Y otra nueva definición:

```
: SACA
CR
." SE EJECUTA LA PALABRA SACA"
;
```

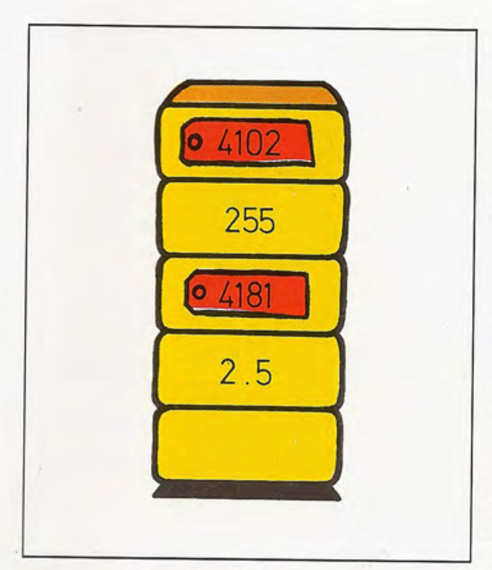
Al ejecutar la primera de las palabras definidas:

ENTRADA <CR>

se obtendrá la siguiente respuesta en la pantalla:



El buffer de entrada actúa como intermediario entre el usuario y el ordenador.



La palabra NUMBER acepta números de buffer de entrada y los deposita en la pila, situando encima de ellos una etiqueta que señala si se trata de un número en punto fijo o de punto flotante.

```
INTRODUCE LO QUE DESEES
```

La respuesta del usuario puede ser la siguiente:

```
SACA <CR>
```

Con lo que el ordenador responderá:

```
ENTRADA
INTRODUCE LO QUE DESEES
SE EJECUTA LA PALABRA SACA
YA SE HA EJECUTADO OK
```

Se observará, pues, que la palabra SACA ha sido ejecutada en medio del proceso de la palabra ENTRADA.

NOTA: A partir de ahora, los listados aparecerán sin figurar en ellos los retornos de carro (<CR>) a introducir como fin de línea.

Introduciendo números

La palabra NUMBER se utiliza para introducir números en la pila desde el buffer de entrada. NUMBER analizará el contenido del buffer de entrada; si no encuentra ningún número al principio del mismo, entenderá que se trata de un cero. El número en cuestión es tomado del buffer de entrada y depositado en la pila, colocando además un código encima del número; el código servirá para precisar si el número es entero (código 4102) o de coma flotante (código 4181).

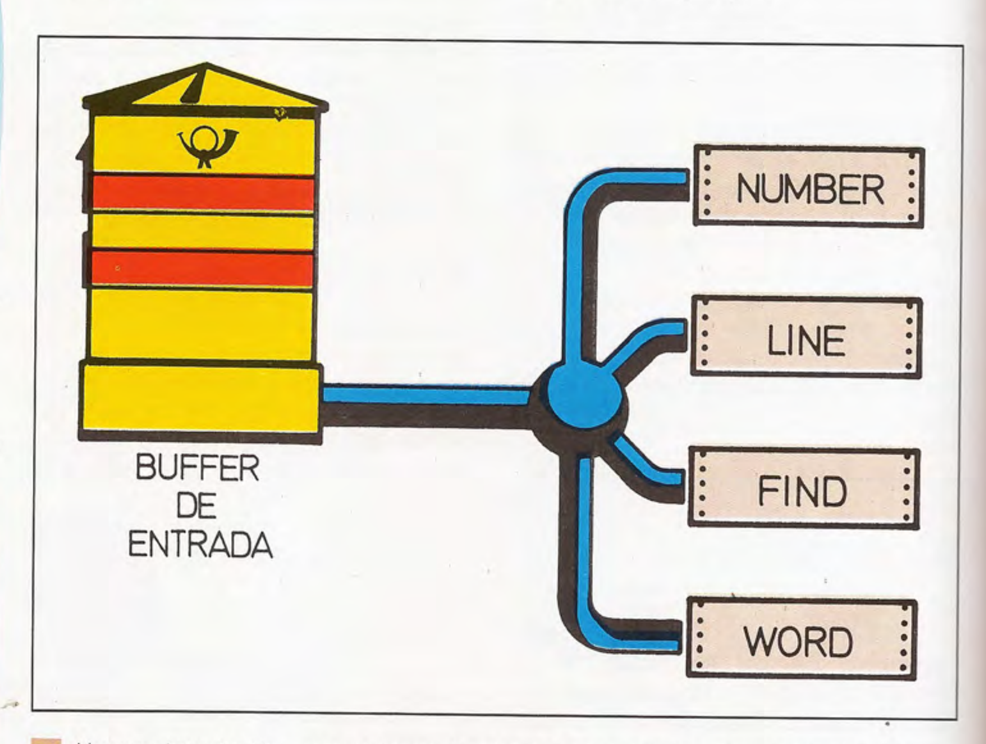
A título de ejemplo, se define a continuación una palabra adecuada para introducir números enteros en la pila, dando además la posibilidad de editar el número incorrecto que pudiera haberse introducido:

```
: ENTERO
BEGIN
RETYPE NUMBER DUP 4181=
IF
." INCORRECTO"
DROP DROP DROP 0
THEN
UNTIL
;
```

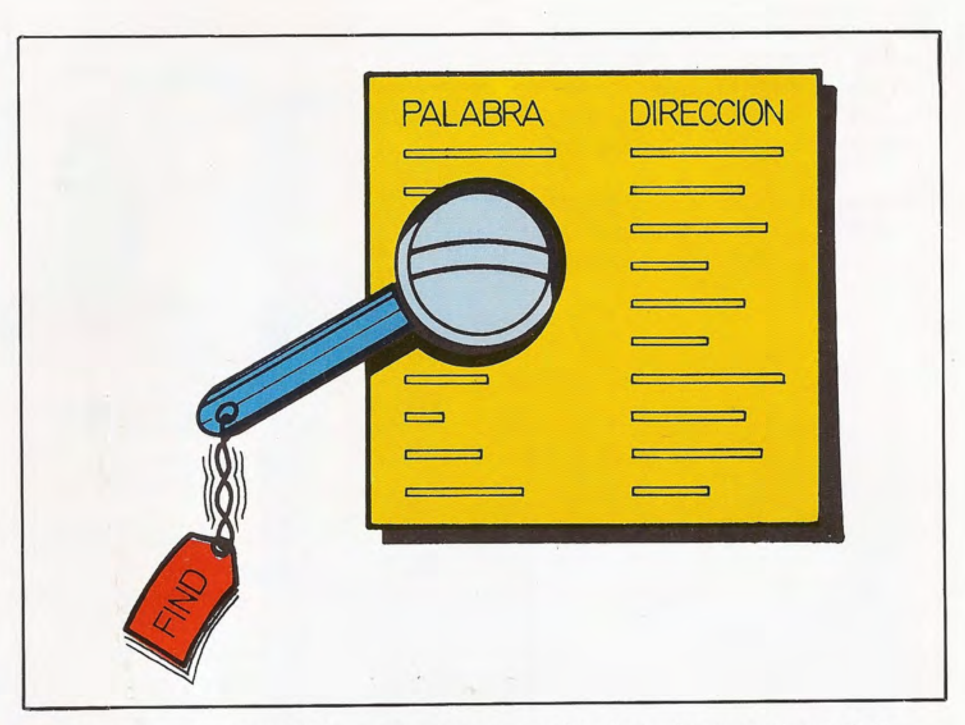
Más sobre palabras

La palabra FIND genera un resultado muy distinto respecto al obtenido con las dos anteriores. FIND toma una palabra del buffer de entrada, y la busca en el diccionario; en el caso de encontrarla, deposita en la cima de la pila la dirección de comienzo de dicha palabra. Esta dirección resulta muy útil en orden al posterior uso de la palabra EXECUTE, la cual ejecutará la palabra cuya dirección de comienzo se encuentre en la cima de la pila.

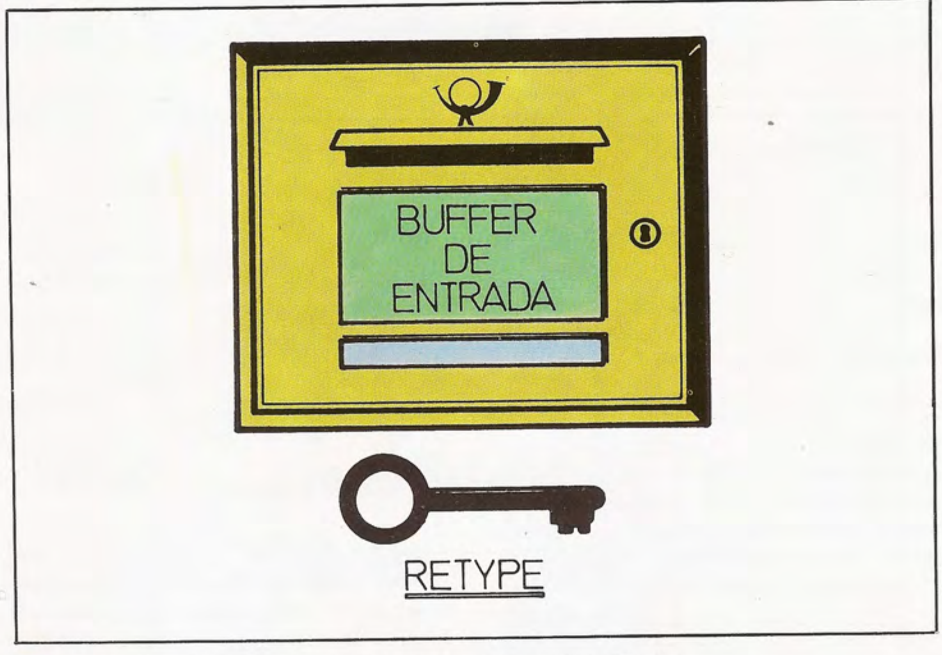
Veamos un ejemplo:



Una vez introducida una secuencia de caracteres en el buffer de entrada, ésta es analizada y recibe un tratamiento distinto dependiendo de la instrucción que se utilice para actualizar el buffer.



Al ejecutar la orden FIND, el ordenador buscará la dirección de comienzo de la palabra que se le indique a través del buffer de entrada, y trasladará dicha palabra a la cima de la pila.



La palabra RETYPE es la llave que permite acceder al buffer de entrada para modificar su contenido.

```
: DIRECCION
QUERY FIND
.
;
```

Al ejecutarlo:

```
DIRECCION <CR>
VLIST <CR>
```

La pantalla mostrará el siguiente resultado:

```
DIRECCION 1581 OK
```

A partir de esta dirección es posible ya ejecutar la referida palabra mediante la orden EXECUTE:

```
1581 EXECUTE <CR>
```

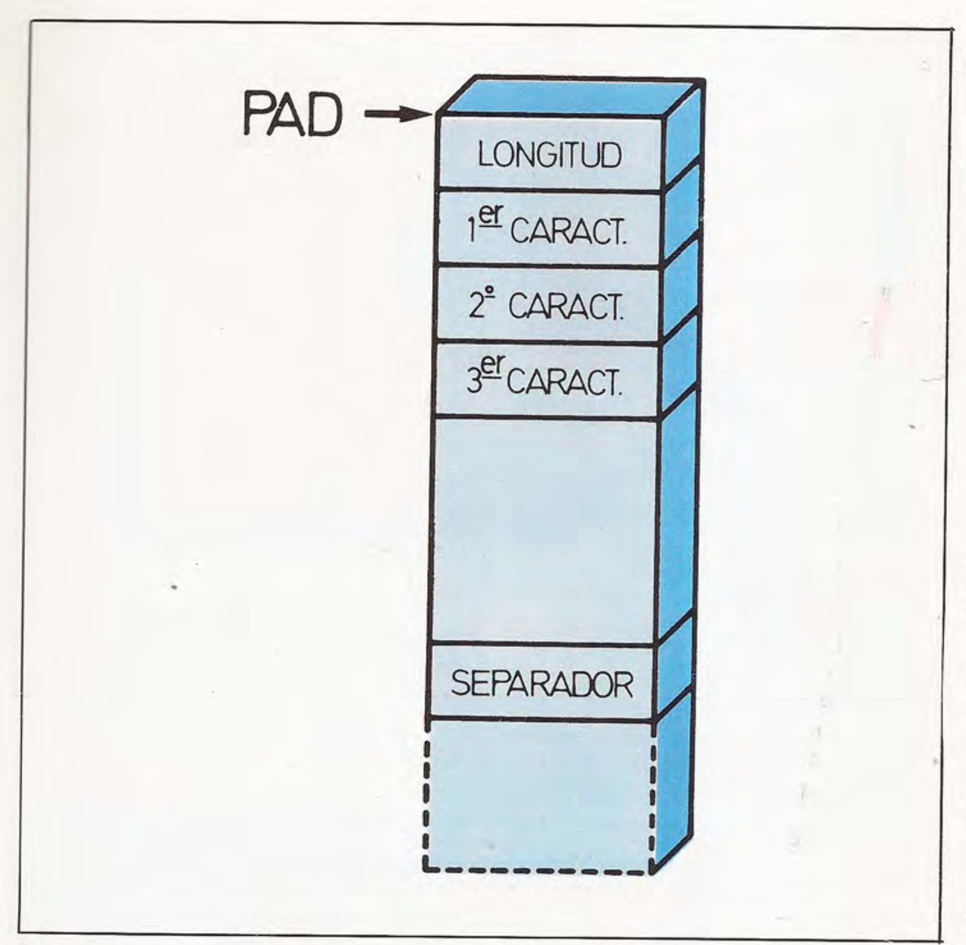
y de inmediato se obtendrá una lista por pantalla de todas las palabras definidas en el diccionario FORTH.

Secuencias de caracteres que no son palabras

Otra palabra FORTH de indudable interés es WORD. Su cometido se concreta en almacenar en la memoria del aparato algunas secuencias de caracteres que no constituyen palabras ni números. La zona de memoria donde se almacenan estos datos recibe el nombre de PAD. En ella, los caracteres son memorizados mediante su correspondiente código ASCII.

Para conocer la dirección de comienzo de esta zona de memoria se puede emplear la palabra PAD, la cual transferirá la dirección de comienzo a la cima de la pila.

La secuencia de acciones asociada a



Representación gráfica de la estructura del PAD.

la ejecución de la palabra WORD es la siguiente:

1. Rellena el PAD con espacios, eliminando cualquier contenido previo.

2. Toma el código ASCII del carácter a emplear como delimitador de entradas, extrayéndolo de la cima de la pila (por lo general, suele coincidir con el espacio cuyo código es 32). El delimitador sirve al ordenador para indicar hasta dónde debe leer del buffer de entrada.

3. Lee del buffer de entrada: obvia los posibles delimitadores que se encuentren al principio del buffer, y lee la palabra hasta el siguiente limitador.

4. Copia en el PAD la palabra leída; coloca al principio la longitud de la misma, a continuación los códigos ASCII de cada uno de los caracteres, y al final el código ASCII de un delimitador (o un cero si no encontró ningún limitador antaes del final del buffer).

5. Coloca en la cima de la pila la dirección de comienzo del PAD.

Lectura directa del teclado

El lenguaje FORTH cuenta con una palabra capaz de leer cualquier pulsación que se efectúe sobre el teclado; se trata de la palabra INKEY. Esta identifica la tecla accionada y deposita su correspondiente código ASCII a la cima de la pila.

Un pequeño inconveniente de esta palabra es que no aguarda a que se pulse una tecla para continuar el proceso, sino que lee del teclado y si no se ha pulsado ninguna tecla continúa el proceso de inmediato. Para solventarlo, es conveniente utilizar la palabra INKEY dentro de un bucle como el siguiente:

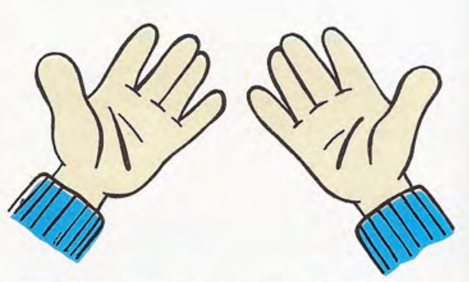
```
: BUCLE
BEGIN
INKEY
UNTIL
;
```

Cuando se ejecute la palabra BUCLE, el ordenador aguardará hasta que se pulse una tecla para continuar el proceso.

Al concluir la ejecución, quedará en la cima de la pila el código ASCII del carácter pulsado, listo para su posterior tratamiento.

Bases de numeración

El sistema de numeración casi generalizado en el ámbito del ser humano es el decimal o de base diez. Ello parece derivar del hecho que la forma más sencilla de contar no es otra que utilizar los dedos: y como quiera que son diez los dedos que se tienen entre las dos manos, el sistema de numeración más obvio y natural es el de base diez.



El sistema de numeración más tradicional y difundido es el decimal o de base diez. Probablemente, su uso generalizado tiene mucho que ver con el hecho de que el hombre posea diez dedos entre ambas manos.

Desde luego pueden emplearse otros sistemas de numeración de distinta base. Tal y como se ha detallado a lo largo de la obra, en el terreno de la informática se utilizan los sistemas de numeración binario, octal y hexadecimal, cuyas bases son, respectivamente: 2, 8 y 16.

Un artículo que puede ser de gran ayuda para comprender las bases de numeración es el ábaco.

Trabajar en un determinado sistema de numeración es muy semejante a operar con un ábaco cuyo número de bolas en cada varilla sea igual a la base menos una.

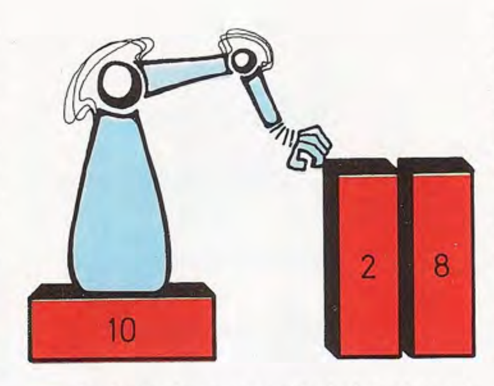
Hemos señalado que los sistemas de numeración más utilizados son aquellos cuyas bases coinciden con 2, 8, 10 y 16. Las razones que motivan su empleo son diversas. En primera instancia, la base decimal es la más obvia en el ámbito humano. A su vez, la binaria es la que emplean internamente los ordenadores. Y por último, el uso de la octal y la hexadecimal se fundamenta en que son muy adecuadas para realizar conversiones entre las bases decimal y binaria; de ahí su frecuente presencia en el mundo de la informática.

Trabajando con otras bases en FORTH

Se ha dicho ya que los ordenadores trabajan internamente en el sistema binario; no obstante están preparados para aceptar números en otras bases de numeración, bases que suelen coincidir generalmente con la decimal, octal y hexadecimal. El lenguaje FORTH contempla tal diversidad de sistemas de numeración, hasta el punto de que permite trabajar en cualquier base de numeración que desee el usuario; basta tan sólo con comunicarlo al ordenador de la forma adecuada.

Normalmente, el FORTH trabaja recibiendo y presentando datos en formato decimal; ésta es la base que se asume al conectar el aparato bajo el control del FORTH.

Para comprobar con qué base se está trabajando en cada momento, es necesario acceder a una variable del sistema que, precisamente, recibe el nombre



La base de numeración habitual en el FORTH es la decimal, si bien este lenguaje está capacitado para operar con otras bases de numeración.

de BASE. Esta variable es de ocho bits, por lo que no es posible trabajar con ella mediante las palabras @ y ! Ello supone que para acceder a la misma hay que recurrir a dos palabras que también operan con direcciones de memoria —al igual que @ y !— aunque accediendo tan sólo a un byte; estas son: C @ y C!

La actuación de las palabras BASE C! y C @ es la que se detalla a continuación.

La palabra BASE deposita en la pila la dirección de memoria correspondiente a la variable que comparte dicho nombre; a continuación, las palabras C! y C@ efectuarán las operaciones que se ordenen a partir de esa dirección de memoria.

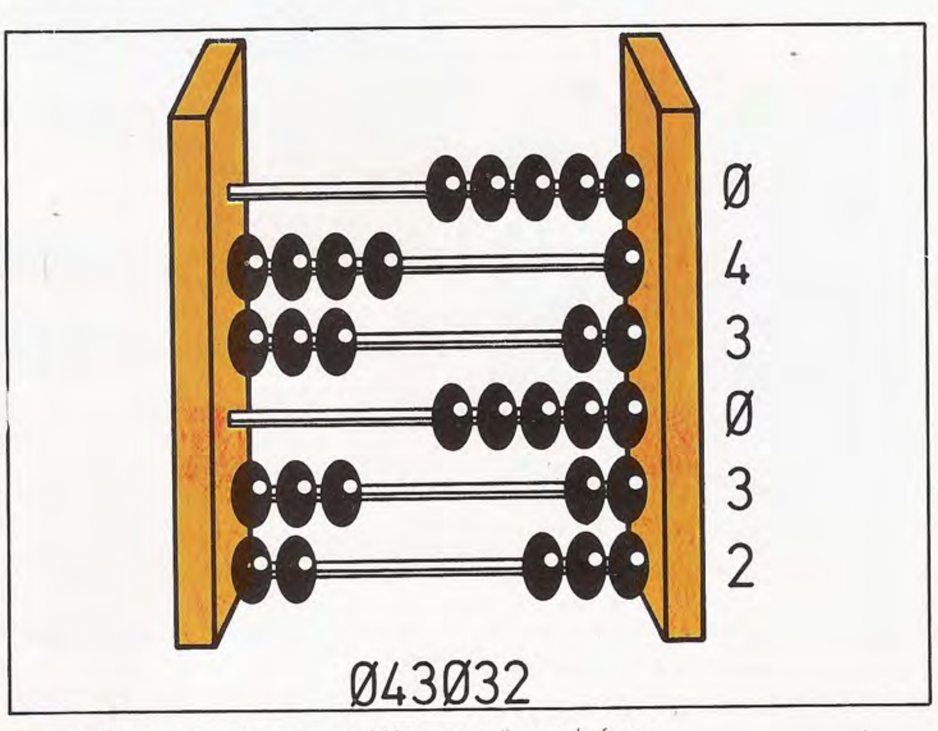
Por ejemplo, si se desea conocer la base con la que se trabaja, puede ejecutarse la siguiente orden:

```
BASE C@ . <CR>
```

La respuesta de la máquina será:

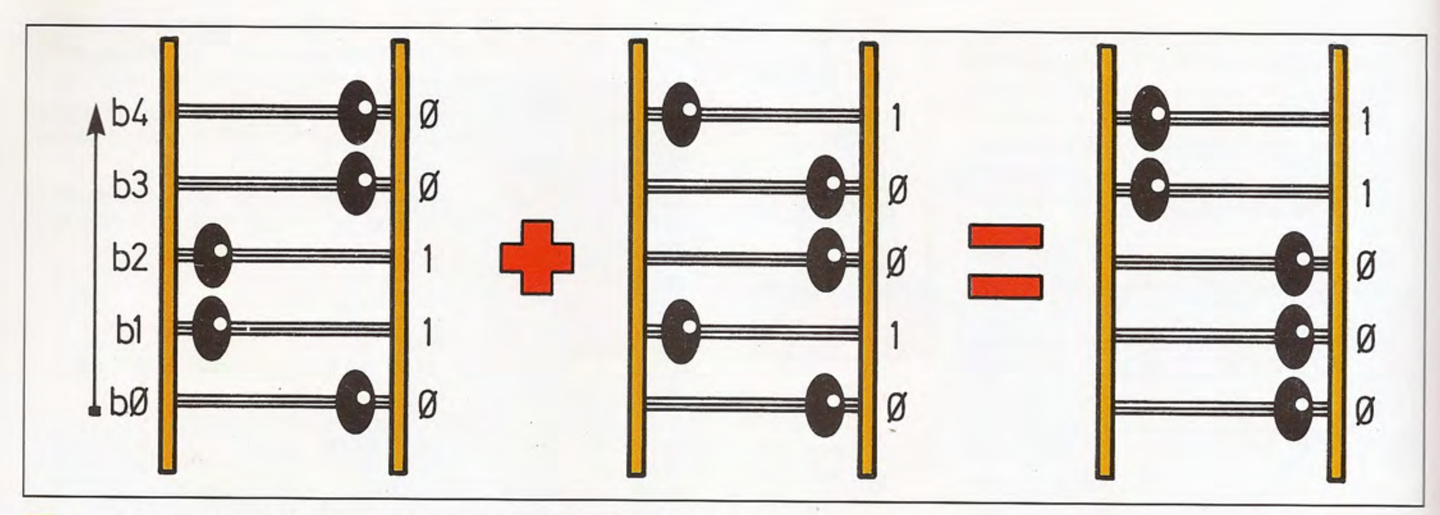


lo cual señala que se está trabajando con el sistema de numeración decimal (base 10).





El ábaco es una herramienta idónea para ilustrar la forma de contar en un sistema de numeración distinto del decimal. El ábaco de la figura permite contar en base seis.

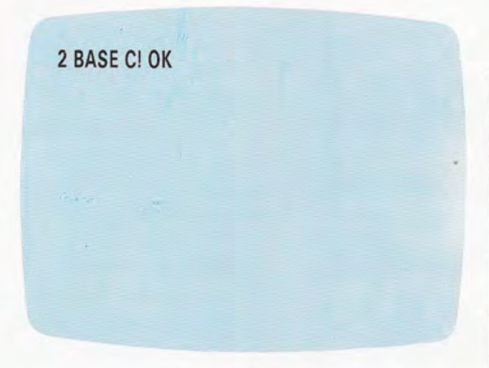


Representación mediante un ábaco de la suma de dos números de forma binaria.

Para cambiar de base, tan sólo es necesario emplear la palabra C! de la siguiente forma:

```
2 BASE C! <CR>
```

Mediante esta configuración de palabras se instruye al ordenador para que cambie el sistema de numeración en curso, de base 10, pasando a base 2. La pantalla mostrará la siguiente respuesta:

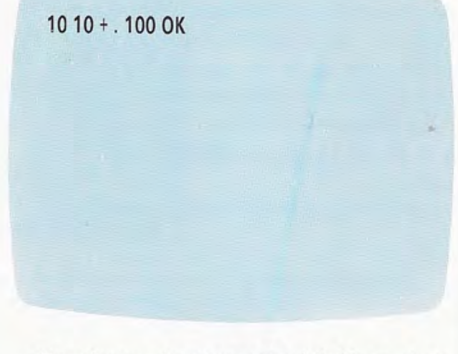


Acto seguido pueden ya efectuarse las operaciones que se deseen en base 2; por ejemplo, una sencilla suma de 2 más 2 unidades:

```
10 10 + . <CR>
```

(Cabe recordar que en el sistema binario, el número decimal 2 se representa como 10).

La respuesta en pantalla coincidirá con la que sigue:

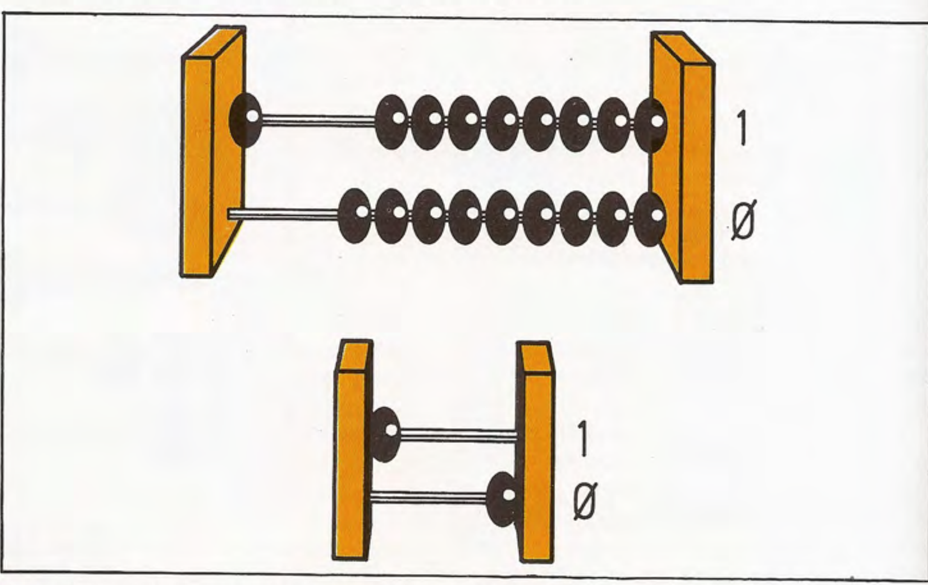


En efecto, la suma de 10 (2 en decimal) más 10 (2 en decimal) es el número binario 100 (el cuatro en el sistema de numeración decimal).

Suponga que ahora deseamos conocer cuál es la base con la que estamos trabajando. Para ello, habrá que repetir el mismo proceso realizado anteriormente:

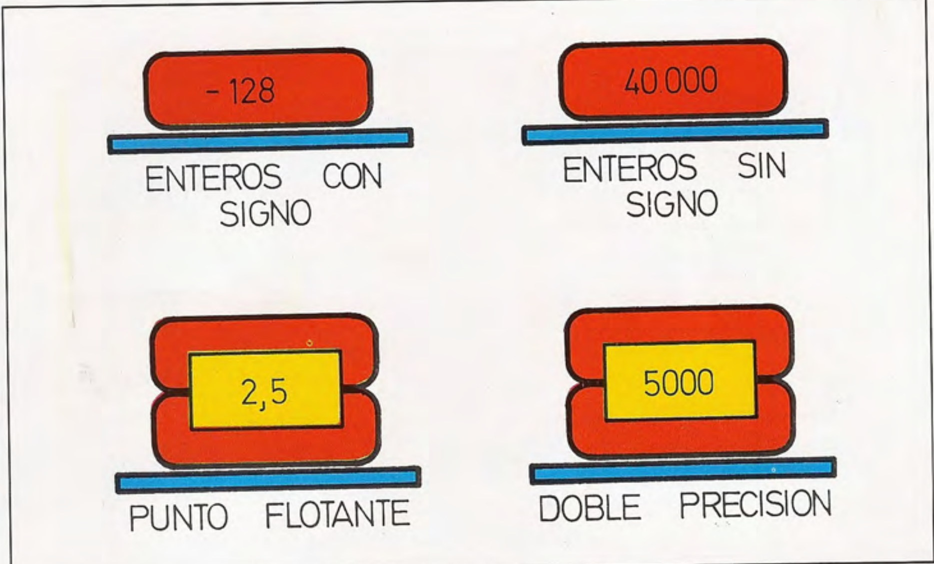
```
BASE C @ . <CR>
```

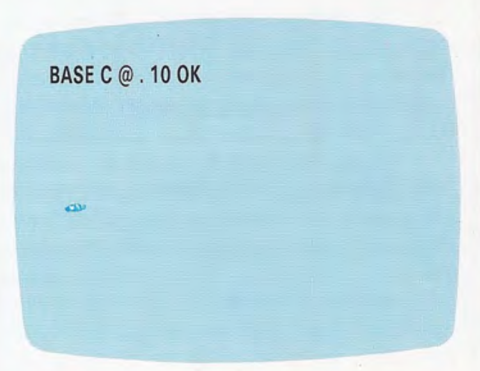
Y la respuesta —tal vez sorprendente— será la que aparece en la siguiente pantalla:



Aunque la representación de dos números en distintas bases de numeración coincida, ello no significa que su valor sea el mismo. Por ejemplo, el número 10 en el sistema binario no coincide con el diez decimal, sino con el valor decimal 2.

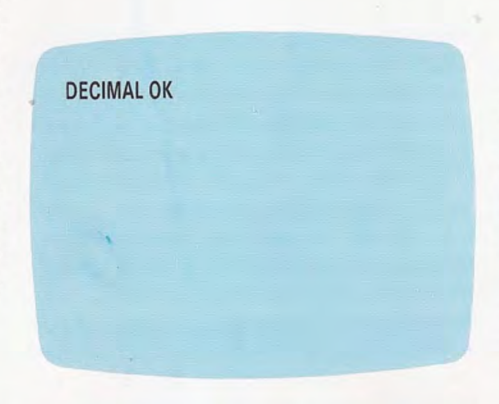
Distintos tipos de numeración utilizables en FORTH y su respectiva ocupación de elementos de la pila. Cabe recordar que cada elemento de la pila ocupa dos bytes de memoria





¿Qué ha sucedido? Nada anormal, puesto que estamos trabajando en base dos; y no hay que olvidar que el número dos se representa en binario como 10. Evidentemente, ésta puede ser una fuente de problemas, ya que el usuario no tiene por qué saber en qué base opera en un determinado momento. Para evitar tal inconveniente, resulta muy oportuno volver al sistema decimal, mediante la palabra DECIMAL, antes de solicitar al ordenador que muestre la base en la que se trabaja.

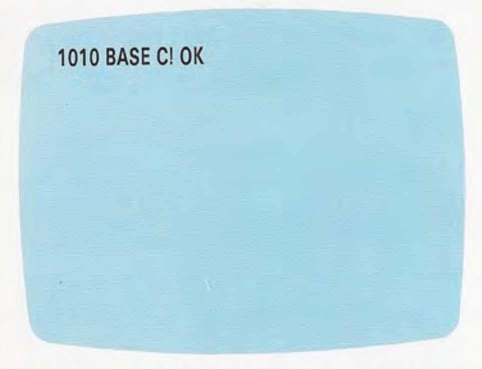
DECIMAL <CR>



Con ello, se tendrá la garantía de haber vuelto al sistema de numeración decimal. También desde el propio sistema binario se podría haber ordenado el retorno a decimal mediante las siguientes palabras:

1010 BASE C! <CR>

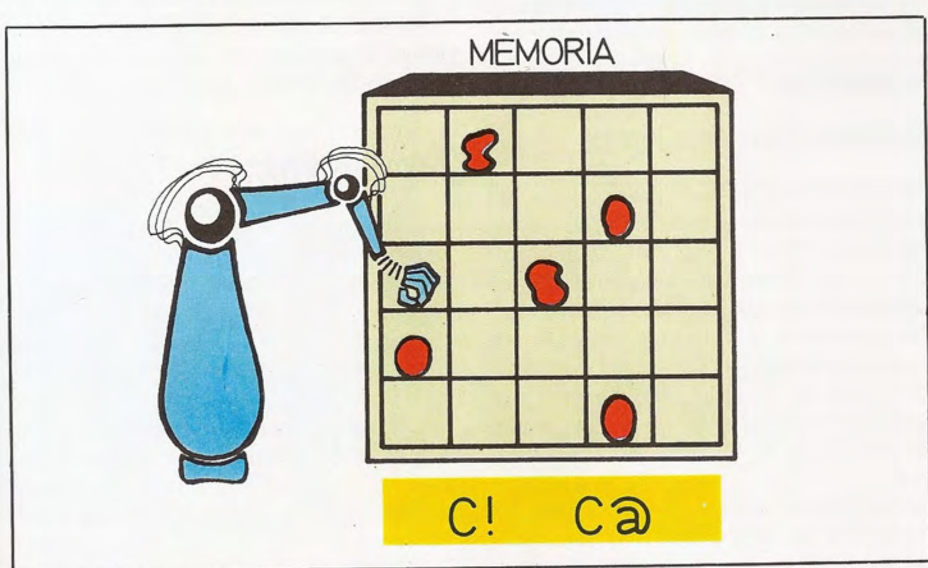
(1010 en binario, corresponde al número 10 decimal). A lo que el ordenador respondería con el siguiente mensaje:



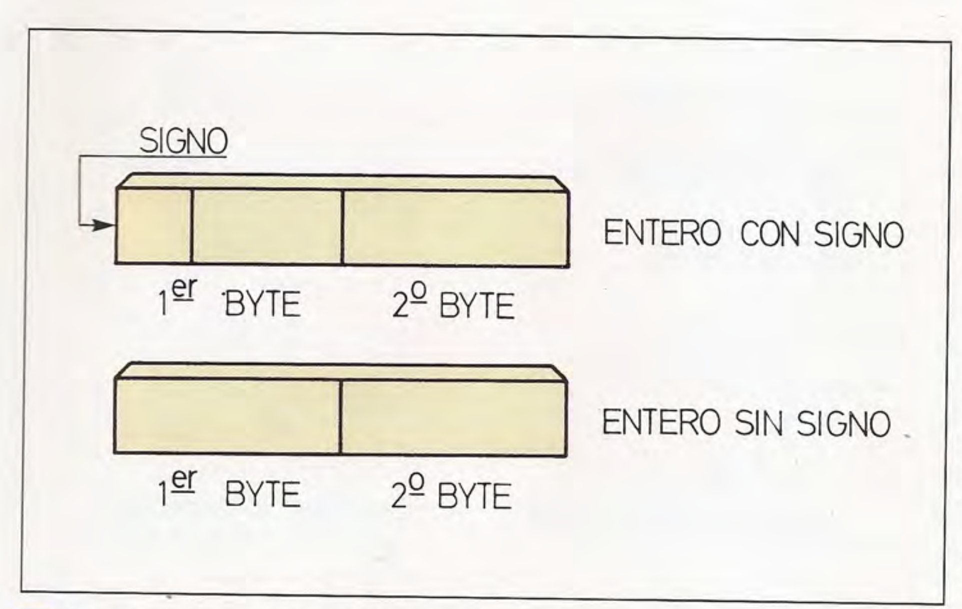
En este punto se podría seleccionar otro sistema de numeración, el hexadecimal, de gran utilidad en determinadas ocasiones:

16 BASE C! <CR>

La imagen en pantalla mostrará el siguiente aspecto:



Las palabras FORTH C@ y C! permiten, respectivamente, leer y cambiar el contenido de las posiciones de memoria.



Representación gráfica de la forma de almacenamiento de los números enteros con y sin signo.



Acto seguido pueden ya efectuarse algunas acciones que revelarán la activación del sistema de numeración hexadecimal. Por ejemplo:

1A 122 + . <CR>



Los dígitos utilizados en el sistema de numeración de base 16 coinciden con las cifras decimales de 0 a 9, además de con las letras A a la F para representar, respectivamente, los valores 10, 11, 12, 13, 14 y 15.

Es obvio pues, que el lenguaje FORTH permite al usuario trabajar a voluntad con cualquier sistema de numeración.

En este caso, hay que tener en cuenta que la base seleccionada sólo afecta a lo que se muestra en la pantalla; internamente, el ordenador recoge los números en una base, los convierte al sistema binario, los opera, y cuando es necesario mostrar el resultado por pantalla, recoge los datos de memoria en binario y los visualiza previa conversión al sistema de numeración seleccionado por el usuario.

Distintos tipos de números

Hasta ahora se han manipulado tan sólo números de un mismo tipo; números enteros con signo y de simple precisión. Sin lugar a dudas, el trabajo con ellos resulta más sencillo; si bien, también es cierto que se ven limitados por sus características. Este es el momento de presentar nuevos tipos de números que ampliarán el abanico de posibilidades del FORTH. La tabla adjunta resume las distintas categorías de números normalmente utilizables en FORTH, así como sus características más representativas.

En ella figuran cuatro tipos de números: cada tipo resulta especialmente adecuado para un distinto marco de aplicación. La primera columna de la tabla incluye el nombre que se otorga a cada categoría numérica; nombre que suele ser lo suficientemente representativo como para evidenciar sus características más relevantes.

La columna bytes señala el número de bytes de memoria necesarios para almacenar un número de dicho tipo.

Hay que recordar que cada elemento de la pila ocupa dos bytes de memoria; de ahí que un número entero con signo esté formado tan sólo por un elemento de la pila, mientras que un número de doble precisión lo está por dos elementos.

La columna de rango define el margen dentro del que pueden estar comprendidos los valores de un número, de acuerdo a su correspondiente tipo. En el caso de los números de doble precisión no se especifica su rango, ya que se pueden utilizar de diversas formas (dependiendo, por ejemplo, de si se considera el signo o no) y por lo tanto el rango puede variar.

La notación científica consiste en fraccionar el número en dos partes: un primer número en punto flotante seguido por la letra E y, tras ésta, un número entero con signo correspondiente a la potencia de 10 por la que deseamos multiplicarlo. Esta notación resultará ampliamente conocida por los habituales usuarios de calculadoras.

En apartados sucesivos se estudiarán cada uno de estos tipos, entrando en detalles acerca de las palabras a utilizar para trabajar con ellos.

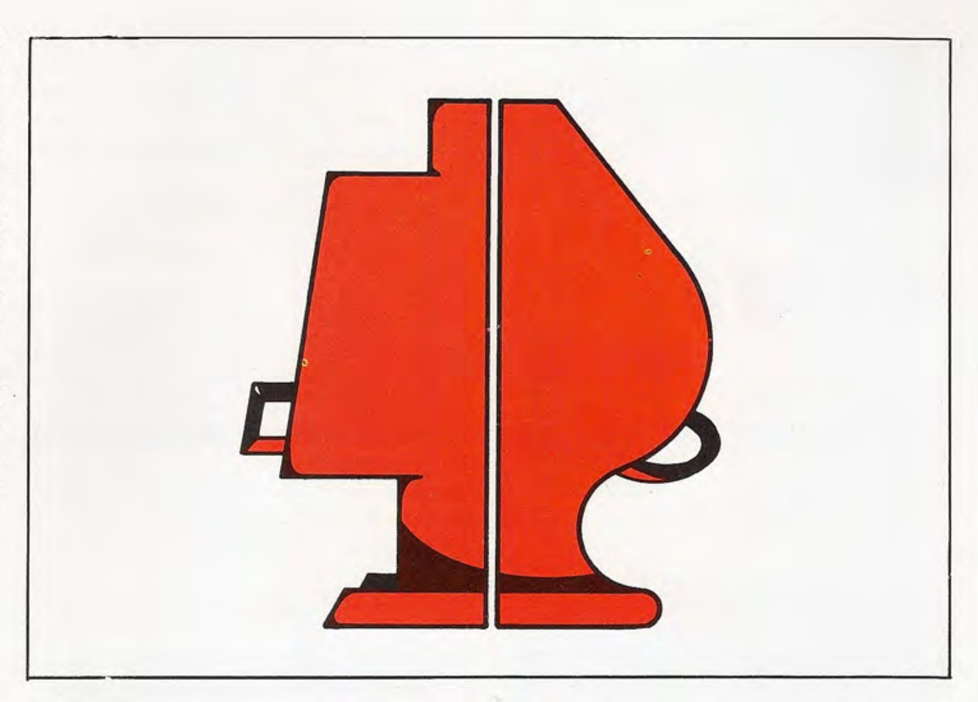
Números enteros

Los números enteros, con o sin signo, se almacenan por medio de dos bytes, lo que se traduce en la posibilidad de representar un total de 65536 combinaciones distintas. En el caso de los números enteros, estas combinaciones han de utilizarse para almacenar tanto los números con signo positivo como sus correspondientes valores negativos. La diferenciación entre unos y otros se establece almacenando los de signo positivo en binario, tal cual, mientras que los negativos se memorizan en complemento a dos.

Tal vez sea conveniente recordar que

la expresión en complemento a dos se obtiene traduciendo el número a binario, complementando a uno su configuración (cambiar los ceros por unos y los unos por los ceros) y sumando una unidad al resultado así obtenido.

La notación en complemento a dos permite saber si un número es positivo o negativo examinando simplemente su primer bit: si es un cero se trata de un número positivo, mientras que si es un uno el número será negativo. Por supuesto, si se prescinde del signo, el bit reservado al efecto quedará libre y, en consecuencia, dicho bit pasará a ocupar la posición de peso superior, permitiendo representar números de mayor magnitud.



Los elementos de la pila parecen tener dos caras: según se interpreten como números enteros con signo o sin signo su apariencia será distinta. Realmente, son más de dos las formas en las que puede ser interpretado un elemento de la pila.

Enteros sin signo

Cuando se trata de trabajar con valores enteros comprendidos dentro del rango que va de 0 a 65535, se puede recurrir al tipo denominado: números enteros sin signo. Este tipo de notación resultará muy útil cuando no se contemple la posibilidad de operar con números negativos, puesto que con los mismos 2 bytes, es posible almacenar números de magnitud muy superior a la que permiten los enteros con signo.

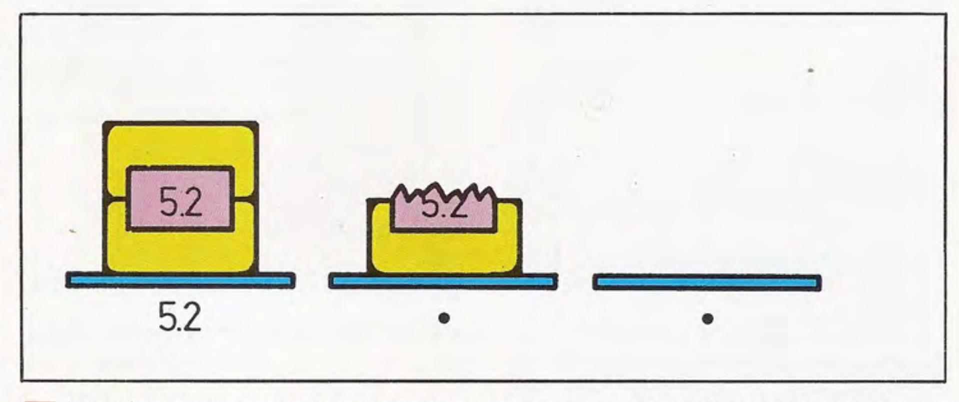
Para visualizar un número entero sin signo es necesario emplear una palabra FORTH cuya denominación se reduce a un solo carácter: U. Veamos su efecto:

```
123 U. <CR>
```

Una vez ejecutada la orden, la pantalla mostrará el siguiente aspecto:

```
123 U. 123 OK
```

Una característica inherente a la pila es que en ella se almacenan los elementos uno encima del otro, sin distinción del tipo numérico asociado al dato que se deposita. Ello significa que el resultado será distinto dependiendo del modo de acceso y de la palabra empleada para visualizar los elementos de la pila.



Los números en punto flotante se almacenan en memoria codificados en cuatro bytes y, en consecuencia, ocupan dos elementos de la pila. Ello significa que para acceder a ellos no basta con tomar un sólo elemento de la misma.

Supongamos que se utiliza la instrucción:

```
-123 U. <CR>
```

Ello equivale a introducir en la pila un número con signo negativo, número que se almacenará en la misma en complemento a dos; y, acto seguido, se ordena su visualización como entero sin signo. La respuesta en pantalla será bastante curiosa:



La zona de exponentes define la correcta situación del punto decimal dentro de un número expresado en punto flotante.

```
-123 U. 65413 OK
```

Desde luego, también cabe realizar la operación contraria:

```
65413. <CR>

65413. -123 OK
```

El resultado de tal experiencia revela que, bajo la perspectiva de la pila, tiene un mismo efecto la introducción de los números 65413 y -123. Por lo tanto, habrá que extremar las precauciones cuando se trabaje con enteros con signo, en orden a evitar que se exceda el rango de validez de estos números; de ocurrir tal situación, se cometerán errores que el ordenador no advertirá.

Un hecho constatable es que la suma de un número en complemento a dos con otro, equivale a la resta de ambos números. Por ejemplo, el resultado de sumar los números 65413 y -123 —uno es el complemento a dos del otro— será igual a cero.

Veamos un nuevo ejemplo:

```
-123 56+DUP . U. <CR>
```

Esta cadena de palabras FORTH ordena a la máquina que calcule la suma de los números -123 y 56, y que muestre el resultado en pantalla como número entero con signo o como entero sin signo; obviamente, para realizar esto último es necesario duplicar previamente la cima de la pila.

Aquí está la respuesta:

```
-123 56+DUP . -67 U. 65469 OK
```

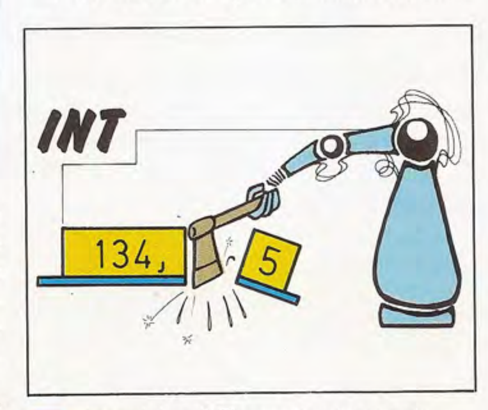
El primer resultado (-67) es el lógico, mientras que el segundo es la consecuencia de acceder a la pila con una palabra inadecuada. Para evitar errores de esta índole, conviene tener presente en todo momento qué tipo de números se encuentran en la pila.

El que sigue es un nuevo ejemplo. Ahora se depositan en la pila dos números enteros sin signo; en este caso, el resultado equívoco se producirá al visualizar el elemento de la pila como número entero sin signo.

```
65413 56+DUP . U. <CR>

65413 56+DUP . -67 U. 65469 OK
```

Cabe observar que las instrucciones aritméticas aplicables a los enteros con signo son también válidas para los números enteros sin signo, a condición de que sean visualizados en pantalla con la orden correcta. Para los enteros sin signo no es válida la palabra de comparación <, si bien, existe otra palabra alternativa perfectamente utilizable: U<.



La palabra INT permite el paso de número en punto flotante a número entero, truncando para ello la parte fraccionaria.

Números en punto flotante

Los números en punto flotante («coma flotante» en el caso español, que recurre a la coma en lugar del punto decimal) se almacenan en memoria mediante cuatro bytes; de ellos, tres se emplean para almacenar un valor numérico que puede ser positivo o negativo. Este número corresponde a las seis cifras significativas o mantisa del número en cuestión, mientras que el cuarto byte se reserva para almacenar la potencia de 10 por la que hay que multiplicar la mantisa para obtener el valor, representado por el conjunto.

La mantisa es un número racional de punto fijo, que se supone con el punto decimal situado a la izquierda de su primera cifra.

Para ilustrar esta idea se muestran seguidamente algunos ejemplos de representación de números en punto flotante:

```
25.23   Mantisa: 0.2523
        Exponente: 2
1234.0  Mantisa: 0.1234
        Exponente: 4
```

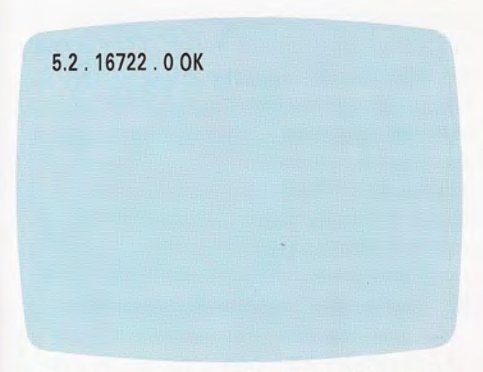
Para trabajar con este tipo de números, el lenguaje FORTH brinda un repertorio de palabras especiales. En primera instancia cabe hablar de la palabra F, destinada a visualizar un número de punto flotante.

Veamos qué sucede cuando se intenta visualizar un número de este tipo depositado en la pila mediante la palabra ".":

```
5.2 .. <CR>
```

(La presencia de dos puntos consecutivos obedece a que los números de punto flotante ocupan dos elementos de la pila).

Ahí está la respuesta:



Por contra, si se utiliza la palabra adecuada, el resultado será bien distinto:



5.2 F. 5.2 OK

Además de la palabra de visualización F, el vocabulario FORTH cuenta con un surtido adicional de palabras destinadas a la ejecución de operaciones aritméticas.

F+ Suma los dos números en punto flotante situados en las posiciones superiores de la pila, y deposita el resultado en la cima de la pila. Así, por ejemplo, el resultado de ejecutar la orden:

5.2 7.1 F+F. <CR>

coincidirá con:

5.2 7.1 F_F. 12.3 OK

F- Resta el segundo número en punto flotante introducido en la pila del introducido en primer lugar; el resultado se deposita en la cima de la pila.

6. 9.4 F-F. <CR>

6. 9.4 F-F. -3.4 OK

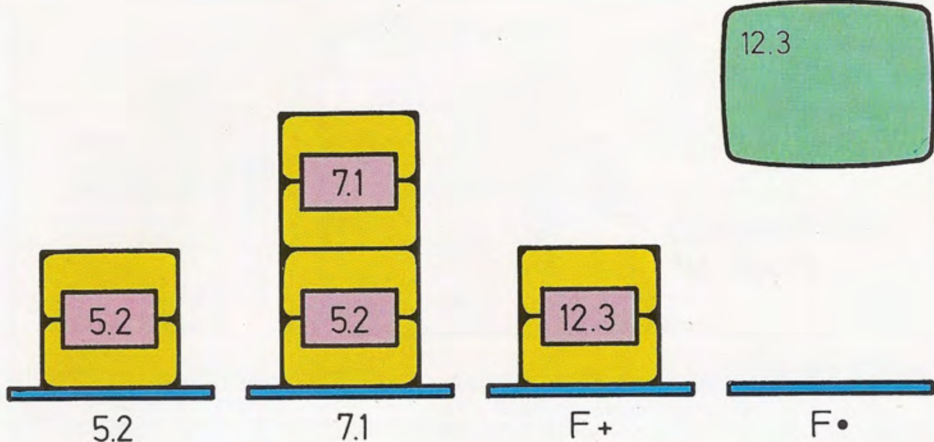
F* Multiplica los dos números de punto flotante situados más cerca de la cima de la pila y deposita el resultado en la misma.

2.1 1. F*F. <CR>

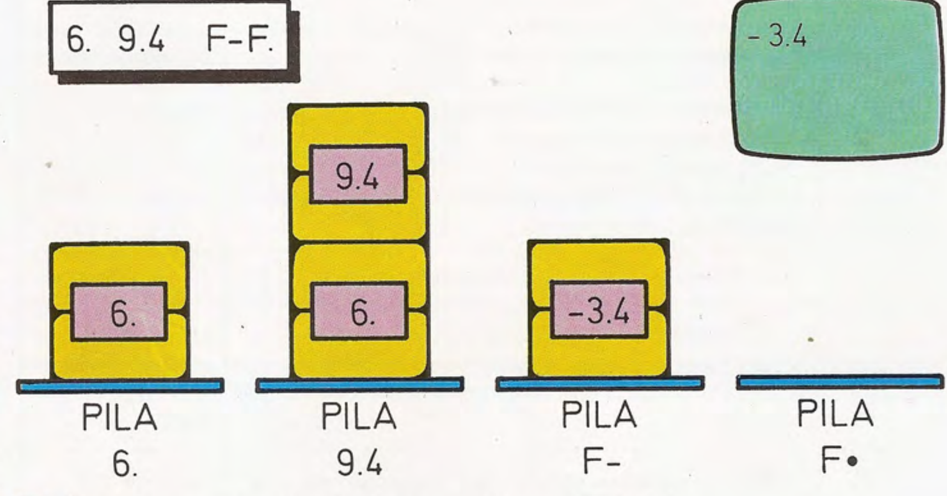
2.1 1. F*F. 2.1 OK

F/ Obtiene la división de los dos números de punto flotante situados en las posiciones superiores de la pila, y vierte el resultado en la cima de la misma.

3.4 5 F/F. <CR>



Evolución de la pila al ejecutar la operación 5.2 + 7.1 y visualizar el resultado.



Manipulaciones de la pila asociadas a la operación 6 - 9.4.

Tipos de números utilizables en FORTH			
TIPO	**BYTES**	**RANGO**	**NOTACION CIENTIFICA**
Enteros con signo	2	-32768 a 32767	No
Enteros sin signo	2	0 a 65535	No
Punto flotante	4	-9.99999E64 a 9.99999E62	Sí
Doble precisión	4	Según su empleo	No

Tabla de órdenes FORTH		
PALABRA	DESCRIPCION	TIPO
QUERY	Permite introducir caracteres en el buffer de entrada.	Palabra de E/S.
RETYPE	Permite editar el contenido del buffer de entrada.	Palabra de E/S.
LINE	Trabaja con el buffer de entrada, aceptándolo como una entrada normal.	Palabra de E/S.
NUMBER	Toma un número del buffer de entrada.	Palabra de E/S.
FIND	Busca la dirección en la que se encuentra compilada la palabra introducida en el buffer de entrada.	Palabra de E/S.
WORD	Trata el contenido del buffer de entrada como un texto, almacenándolo en el PAD.	Palabra de E/S.
EXECUTE	Ejecuta la palabra cuya dirección de memoria se encuentra en la pila.	Manejo de memoria.
PAD	Da la dirección de comienzo del PAD.	Manejo de memoria.
INKEY	Lee el teclado y toma el código ASCII del carácter introducido, depositándolo en la cima de la pila.	Palabra de E/S.
BASE	Contiene la base del sistema de numeración con el que se está trabajando.	Variable del sistema.
C	Lee el contenido de una posición de memoria.	Manejo de memoria.
C!	Cambia el contenido de una posición de memoria.	Manejo de memoria.
DECIMAL	Restaura el valor de la base del sistema de numeración utilizado a decimal.	Sistema de numeración.
U.	Muestra por pantalla un número entero sin signo extraído de la cima de la pila.	Palabra de E/S.
F.	Visualiza un número de punto flotante constituido por los dos elementos más cercanos a la cima de la pila.	Palabra de E/S.
F+	Suma dos números de punto flotante, tomando los elementos más próximos a la cima de la pila.	Palabra aritmética.
F-	Resta los dos números de punto flotante que se encuentran en las posiciones superiores de la pila.	Palabra aritmética.
F*	Multiplica dos números de punto flotante.	Palabra aritmética.
F/	Divide dos números de punto flotante.	Palabra aritmética.
FNEGATE	Cambia el signo de un número de punto flotante.	Palabra aritmética.
INT	Convierte un número de punto flotante a número entero, truncando la parte fraccionaria.	Conversión de tipo.
UFLOAT	Transforma un número entero en número de punto flotante.	Conversión de tipo.

3.4 5. F/F. .68 OK

FNEGATE Cambia el signo del número en punto flotante que se encuentre en la posición superior de la pila.

12. FNEGATE F. <CR>

12. FNEGATE F. -12. OK

Todas las operaciones descritas actuarán tomando los elementos que se encuentren más cerca de la cima de la pila y considerándolos como números de punto flotante. Esta es una precisión muy a tener en cuenta en el caso de ejecutar operaciones aritméticas sobre números de distinto tipo.

Los números en notación científica son también de punto flotante, tal y como muestra el siguiente ejemplo:

4.7E1 4.7E3 F+F. <CR>

4.7E1 4.7E3 F+F. 4747. OK

Conversión de números de un tipo a otro

Existen palabras FORTH que permiten el paso de números en punto flotante a números enteros y viceversa; éstas son las palabras INT y UFLOAT.

La primera de ellas convierte un número de punto flotante en entero, truncando la parte fraccionaria:

5. 2. F/INT . <CR>

3. 2. F/INT . 2 OK

A su vez, UFLOAT transforma un número entero en número de punto flotante:

2 UFLOAT 2.3 F+F. <CR>

2 UFLOAT 2.3 F+F. 4.3 OK

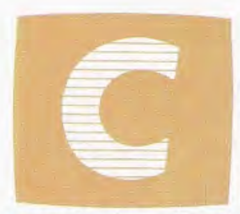
Esta última palabra presenta ciertos problemas, conducentes a error, cuando actúa sobre números negativos; por ejemplo:

-12 UFLOAT F. <CR>

-12 UFLOAT F. 65524 OK

FORTH (y 4)

Números de doble precisión, arrays y últimos detalles



Cuando es necesaria una mayor precisión en los cálculos a realizar en FORTH, se puede recurrir al empleo de variables de doble precisión. Para representar un número de doble precisión hay que contar con dos elementos de pila; ello significa que un número de esta categoría se almacena en cuatro bytes de memoria (32 bits).

Estos números son relativamente difíciles de manejar. Si bien existen instrucciones adecuadas para realizar algunas operaciones con ellos, no es posible trasladarlos directamente a la pila; por lo demás, no existen en todas las versiones de FORTH palabras adecuadas para mostrarlos directamente por pantalla (aunque más tarde se estudiará un método que permite visualizarlos realizando una serie de operaciones con el PAD).

El almacenamiento en memoria de los números de esta índole se realiza en binario y fraccionados en dos partes. la primera coincide con un valor numérico que representa los 16 bits menos significativos del equivalente binario del número a introducir y, tras ésta, se introduce el valor correspondiente a los dieciséis bits más significativos.

Como quiera que en principio, no siempre existe una palabra específica para mostrarlos por pantalla, es preciso recurrir a la palabra "."; ésta sólo permite la visualización de un elemento de la pila, luego habrá que ejecutar dos palabras de este tipo para leer un número de doble precisión.

Para introducir el número 127 en formato de doble precisión, hay que proceder a la siguiente instrucción de datos:

```
127 0 <CR>
```

Una forma más eficiente de trabajar con estos números la proporciona el sistema hexadecimal. Al operar en este sistema, los dos elementos de la pila que se visualizan sí corresponden exactamente a las cifras que conforman el número, aunque en un orden un poco peculiar.

Un número de doble precisión se representa por medio de un número hexadecimal de 8 cifras; para introducirlo en el ordenador hay que empezar por las cuatro cifras situadas más a la derecha (las menos significativas) y, a continuación, se introducirán las cuatro de la izquierda (las más significativas). En el siguiente ejemplo, se asume que el sistema de numeración en el que se trabaja es el hexadecimal.

Suponga que se desea introducir el número:

```
25AC7645
```

Para ello habrá que depositar en la pila los dos números siguientes:

```
7645 25AC <CR>
```

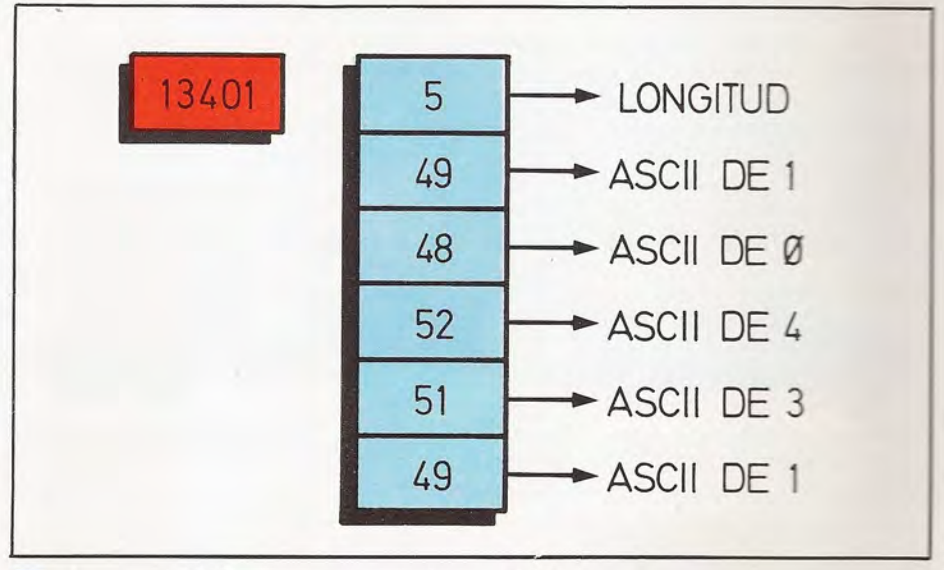
Tras accionar el retorno de carro, el número en cuestión pasará a la memoria interna. Para visualizarlo será preciso ejecutar dos palabras de tipo "."; esto es:

```
.. <CR>
```

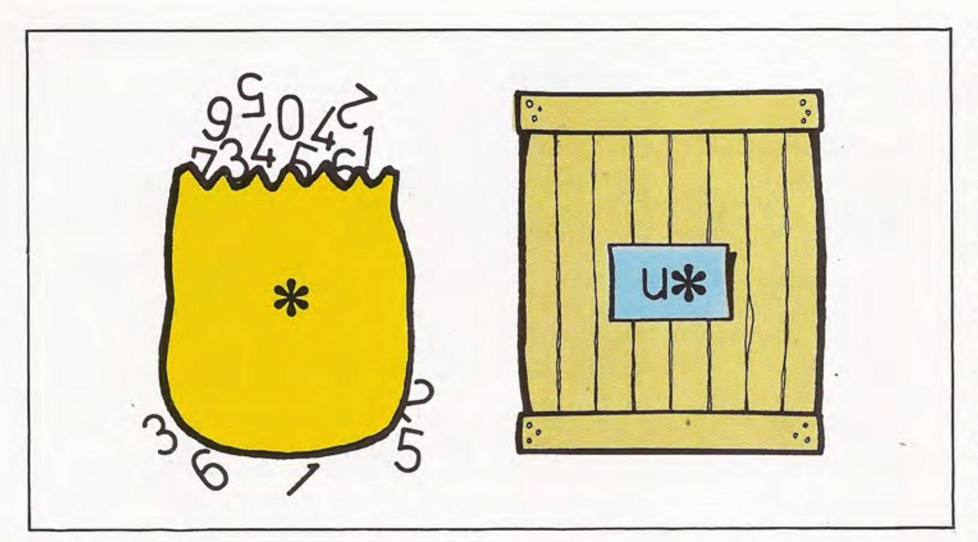
Tras ello, la pantalla mostrará los dos números hexadecimales que fueron introducidos en su momento:



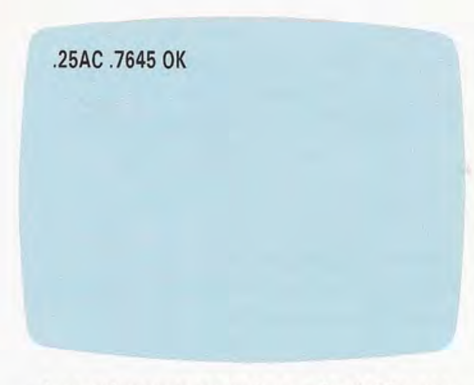
ASCII es una palabra FORTH que busca en la tabla de códigos ASCII el correspondiente al carácter que se especifique, el cual será depositado en la pila.



Aspecto del PAD al almacenar el número 13401.

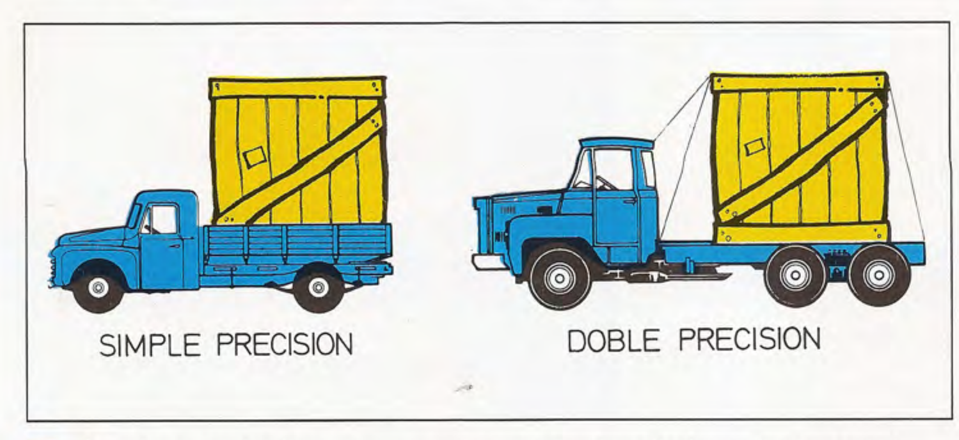


La necesidad de utilizar la palabra U deriva del hecho de que el producto de dos enteros sin signo suele exceder la capacidad de representación de valores de este tipo de números.*



Los números negativos de doble precisión se almacenan en complemento a dos. Normalmente, cuando se trabaja con otros tipos de números, para almacenar un número negativo es suficiente con anteponer el signo, y el propio ordenador se encarga de realizar la complementación y almacenarlo en el formato adecuado. En este caso, como quiera que el ordenador no introduce el número sino que lo hace el propio usuario (quien debe introducirlo como dos números separados), también éste último se verá obligado a tener en cuenta el signo y complementar el número para obtener el resultado adecuado. Así, para introducir el número −234 en formato de doble precisión, es necesario introducir la siguiente secuencia:

```
-234 -1 <CR>
```



Cuando es preciso lograr una mayor exactitud en los cálculos realizados con el lenguaje FORTH, es necesario recurrir a los números de doble precisión.

Operaciones con números de doble precisión

Existen una serie de operaciones en las que intervienen números de doble precisión, ya sea como operadores, constituyendo el resultado, o incluso como ambas cosas. A continuación se exponen las palabras más importantes relacionadas con los números de doble precisión.

D+ Palabra que se emplea para sumar dos números de doble precisión.

Como el ordenador no es capaz de distinguir qué número de los que se encuentran en la pila son de doble precisión y cuáles no, entenderá que lo que debe hacer es tomar los dos elementos situados más cerca de la cima de la pila e interpretarlos como uno de los números y, a continuación, tomar los dos elementos siguientes e interpretarlos del mismo modo. Acto seguido sumará ambos números y depositará al resultado en la cima de la pila, adoptando éste el formato de doble precisión.

Por ejemplo, a la orden:

```
8 0 6 0 D+ . . <CR>
```

el ordenador dará la siguiente respuesta:

```
8 0 6 0 D+ .0 .14 OK
```

DNEGATE Permite cambiar el signo de un número de doble precisión. Al ejecutar la instrucción:

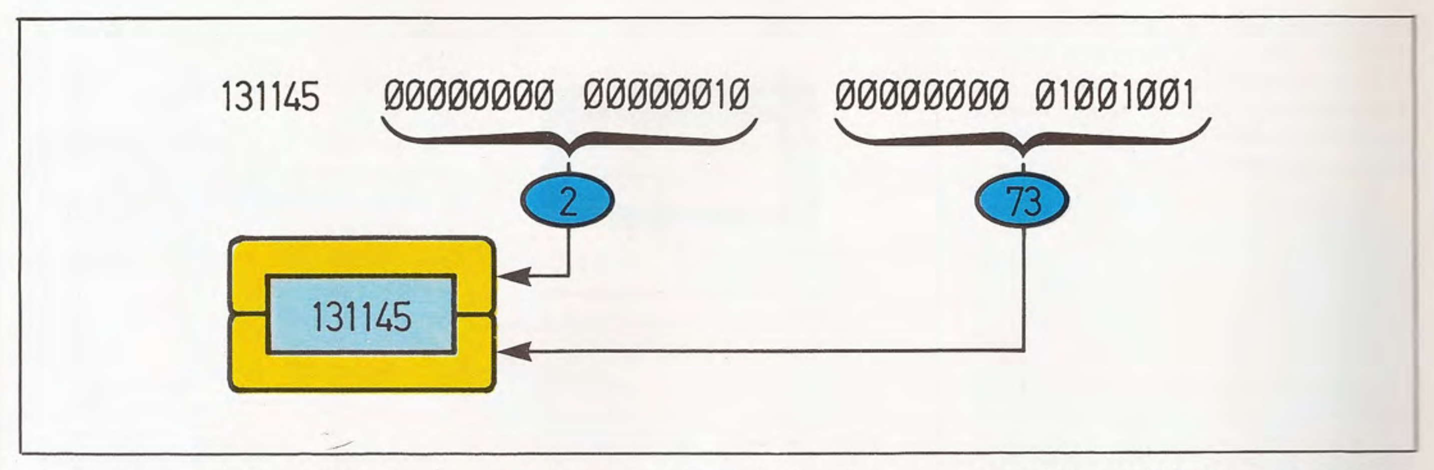
```
3 0 DNEGATE . . <CR>
```

se obtendrá el siguiente resultado en pantalla:

```
3 0 DNEGATE .-1 .-3 OK
```

U* Se trata de una palabra que multiplica dos números; sin embargo, estos dos números no son de doble precisión, sino enteros sin signo. La existencia de esta peculiar orden se debe a que resulta frecuente que el producto de dos números enteros de simple precisión desborde la capacidad de representación de este tipo numérico. Por ejemplo:

```
3 4 U* . . <CR>
```



Para introducir en el ordenador un número en doble precisión, hay que empezar calculando su equivalente binario. Acto seguido, se introducirá en primer lugar el número representativo de los 16 bits menos significativos, y luego el correspondiente a los 16 bits más significativos.

La respuesta del ordenador será:

```
3 4 U*.0 .12 OK
```

D< Palabra FORTH que compara dos números de doble precisión y comprueba si uno de ellos es menor que el otro. Ejemplos:

```
8 0 6 0 D<. <CR>

8 0 6 0 D< .0 OK
```

U/MOD Realiza la división de dos números, entregando el cociente y el módulo de la división. Toma el primer elemento de la pila como entero sin signo, y agrupa los dos siguientes para sintetizar un número de doble precisión. Es una variante de la palabra /MOD que trabaja con enteros sin signo, con la salvedad de que el dividendo es ahora un número de doble longitud.
Ejemplos:

```
13 0 6 U/MOD .. <CR>
13 0 6 U/MOD .2 . 1 OK
```

Impresión de números en pantalla

La filosofía fundamental del formateado de números para su visualización en pantalla consiste en pasar los dígitos al PAD en orden inverso, para más tarde visualizarlo con el formato deseado. Estos números han de ser de doble precisión y positivos.

Existen una serie de palabras FORTH especializadas en el transporte de los números de la pila al PAD; éstas son las siguientes:

<# Empieza a pasar el número al PAD.

#> Termina de pasar el número al PAD; elimina el entero de doble precisión de la pila y coloca en ella la dirección y longitud del PAD.

#→# Extrae un dígito del número y lo lleva al PAD.

#S Extrae todos los dígitos hasta que el siguiente sea cero.

HOLD Guarda un carácter ASCII en el PAD.

ASCII Deposita en la pila el código ASCII del siguiente carácter.
Por ejemplo:

```
ASCII # . <CR>

ASCII # .35 OK
```

A continuación se define una palabra que trabaja adoptando este sistema:

```
: NUM
5 0
DO
CR O < ###
####>TYPE
LOOP
;
```

Su actuación queda patente en la siguiente ejecución:

```
2 3 4 12 12 NUM <CR>

00012
00012
00004
00003
00002 OK
```

Desde luego, esta rutina parte del supuesto de que en la pila se encuentran cinco números que se desean visualizar; si no fuera así, el proceso conduciría a error.

La palabra SIGN permite pasar al PAD

el signo de un número entero residente en la pila. Meidante esta nueva palabra FORTH, se puede crear la palabra "D.", la cual permite visualizar en pantalla un número de doble precisión con su signo correspondiente:

```
: D.
DUP >R DUP 0<
IF
  DNEGATE
THEN
<##S R> SIGN #>
TYPE
;
```

Al ejecutarla:

-12 -1 D. <CR>

se obtiene la siguiente respuesta:

-12 -1 D. -12 OK

Taxonomía de las palabras FORTH

En FORTH existen distintos tipos de palabras. Hasta el momento se han empelado tres tipos: las que se definen mediante la palabra «dos puntos» (:), las constantes que se definen mediante CONSTANT y las variables que se definen por medio de VARIABLE. Existen muchos más tipos de palabras, ya que el usuario puede definir tipos de palabras para su propio uso. Dentro de semejante diversidad caben, por ejemplo, palabras para almacenar cadenas de caracteres y arrays.

Las palabras que componen el diccionario FORTH tienen una serie de puntos comunes:

1. Cada palabra tiene un nombre que sirve para identificarla.
2. Cada palabra está encadenada a las demás. Todas las palabras se basan para su definición en palabras ya existentes.
3. Cada palabra tiene una información (campo de código) que indica cuál ha de ser la acción a realizar cuando se ejecute.



Al operar con números de doble precisión, el propio usuario debe calcular el complemento a dos de los números negativos que hay que intoducir en el ordenador. El diagrama de flujo refleja los pasos necesarios para obtener el complemento a dos de un número.

4. En su definición, las plabras incluyen información adicional (el campo de parámetros).

El nombre, el lazo y el campo de código tienen el mismo formato para todas las palabras y se agrupan en la zona de cabecera.

El contenido de la zona de parámetros varía de uno a otro tipo de palabra. La forma de emplear los referidos parámetros depende del tipo de la palabra, el cual se especifica mediante el campo de código.

Los «arrays» en FORTH

La forma más simple de crear un nuevo tipo de palabra consiste en hacer uso de CREATE. Y, desde luego, es posible construir un campo de parámetros específicos para ella. Su campo de código especificará que cuando esa nueva palabra se ejecute, sea depositada en la pila la dirección de comienzo de su campo de parámetros. De esta forma, el programador puede recoger tal dirección para emplear el campo de parámetros a plena voluntad.

Para construir la cabecera se emplea la palabra CREATE. Suponga que se desea crear un array numérico que almacenará la duración en días de cada uno de los meses del año.

CREATE MESES <CR>

Para construir el campo de parámetros se dispone de la palabra auxiliar ",", cuyo cometido es extraer un número de la pila y guardarlo en el diccionario, reservando para ello dos bytes adicionales que añade al diccionario.

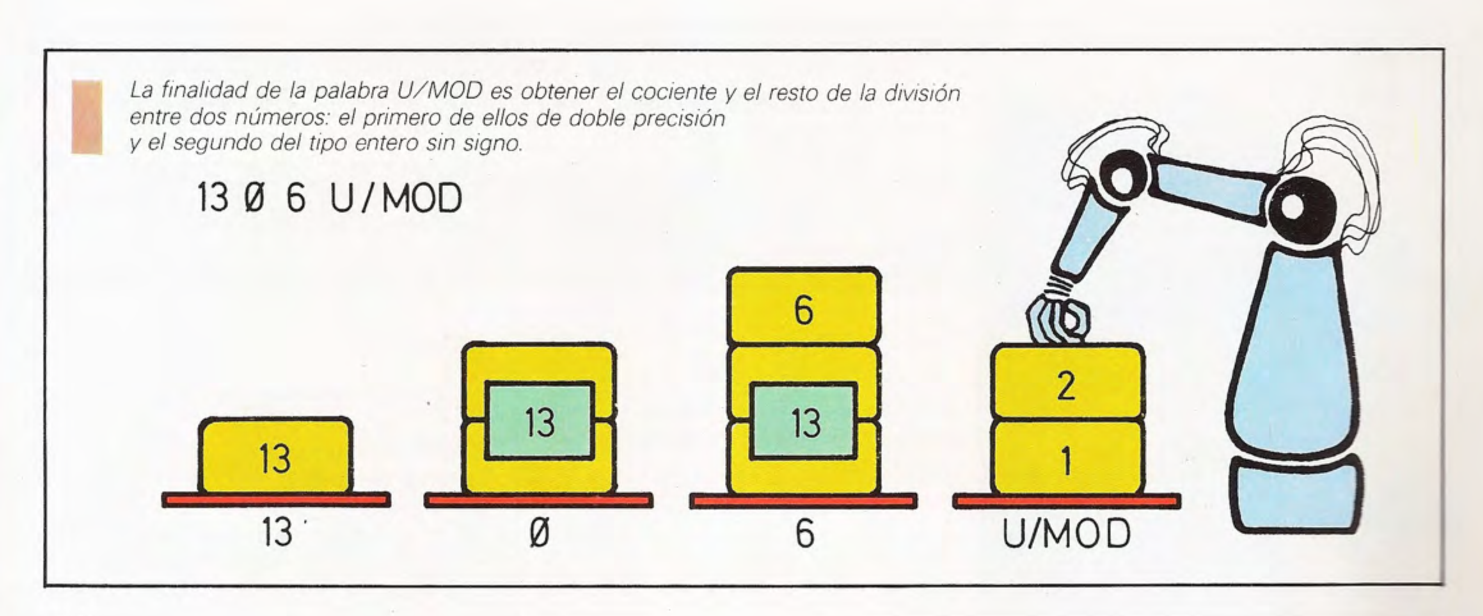
En esencia se trata de ir situando los elementos que formarán el ARRAY en la zona de parámetros de la palabra.

A continuación se muestra la forma de introducir el ejemplo propuesto en el ordenador:

31 ,28 ,31 ,30 ,31 ,30 ,31 ,31 ,30 ,31 ,30 ,31 <CR>

31 ,28 ,31 ,30 ,31 ,30 ,31 ,31 ,30 ,31,
30, 31 OK

Para leer un elemento se va a definir la palabra MES. A la hora de emplearla hay que situar el número del elemento que se desea leer en la pila y, posteriormente, invocar a la mencionada palabra:



La finalidad de la palabra U/MOD es obtener el cociente y el resto de la división entre dos números: el primero de ellos de doble precisión y el segundo del tipo entero sin signo.

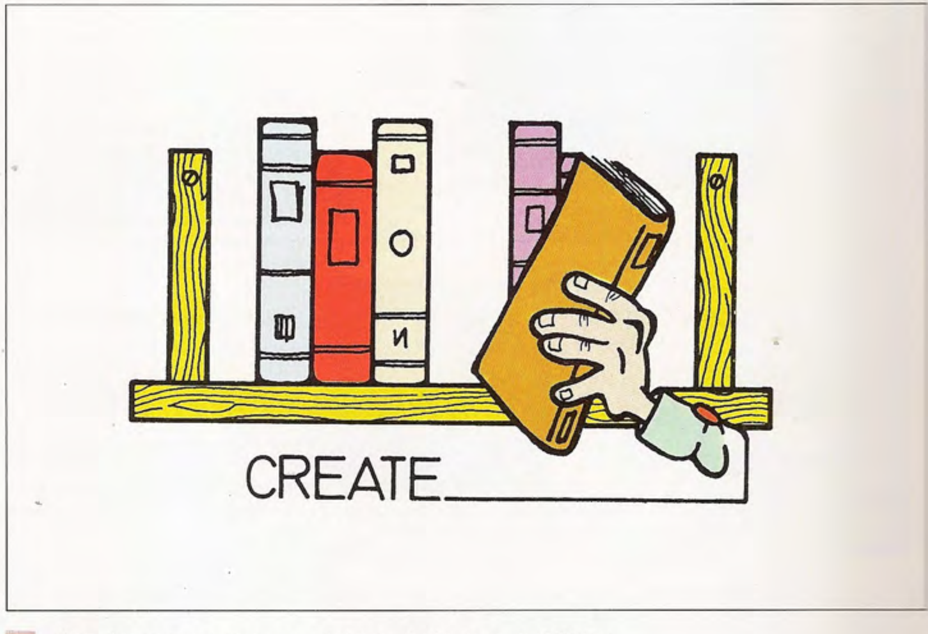
```
. M ES
1- DUP+M ES ES
+
;
```

Suponga que se desea leer el segundo elemento:

```
2 M ES <CR>
```

La respuesta que se obtendrá será la siguiente:

```
2 MES 28 OK
```



Para incorporar nuevas palabras al diccionario FORTH puede utilizarse la palabra clave CREATE.

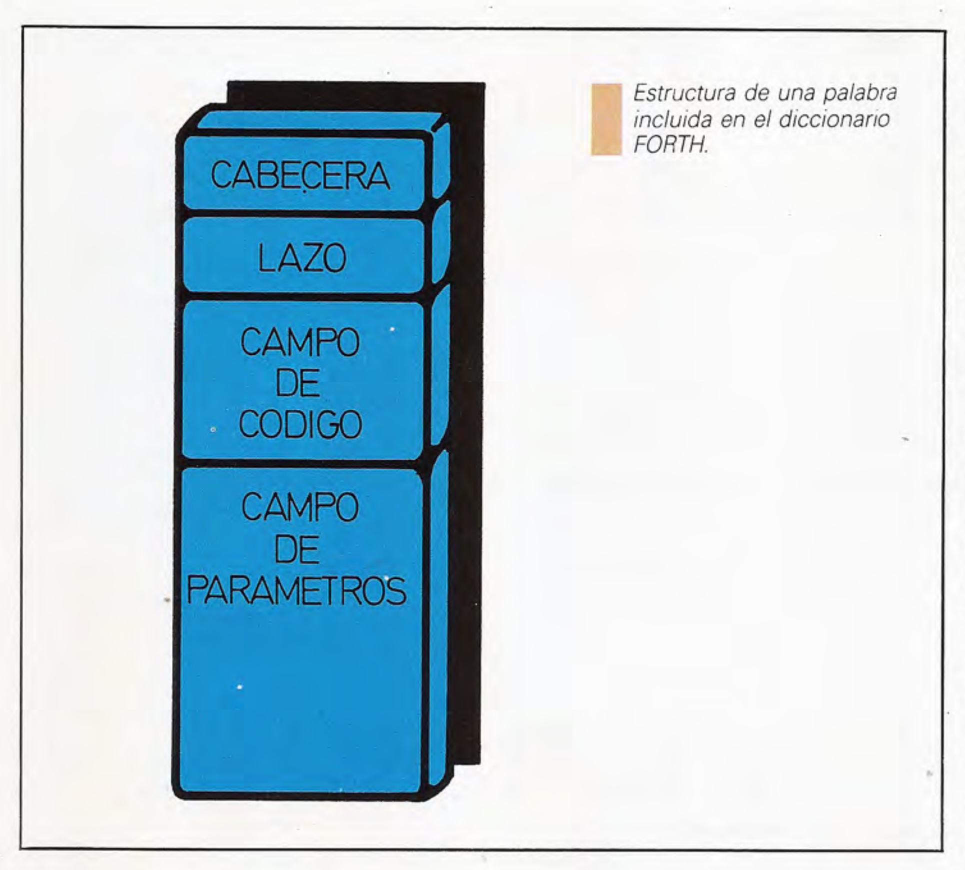
Cadenas de caracteres

Las cadenas de caracteres se almacenan en casi todos los lenguajes como secuencias de códigos ASCII; en FORTH se utiliza el campo de parámetros de una palabra para almacenar los códigos que componen la cadena. Como los códigos ASCII ocupan un solo byte cada uno, es necesario emplear la palabra "C," para tansferir el código de un carácter al campo de parámetros de la palabra. "C," es la versión para un byte de ",": deposita un número en el diccionario, pero emplea para ello un byte.

Por ejemplo, puede utilizarse la palabra ALLOT para reservar un número de bytes para el campo de parámetros.

Para introducir una cadena en una palabra es posible emplear la siguiente palabra FORTH:

```
: CADENA
32 WORD DUP 1+
SWAP C OVER + SWAP
DO
 |C C,
LOOP
;
```



Estructura de una palabra incluida en el diccionario FORTH.

Veamos con detalle cuál es la actuación de cada uno de los elementos que intervienen en la definición de la nueva palabra CADENA:

: Comienzo de la definición de la palabra.

CADENA Nombre identificador de la palabra.

32 Sitúa en la pila el valor 32.

WORD Interpretará el valor 32 que se encuentra en la pila como el código ASCII del carácter que ha de tomar como separador; en este caso se trata de un espacio en blanco. A continuación, WORD toma el contenido del buffer de entrada y lo coloca en el PAD, sitúa la longitud de la palabra al comienzo del PAD y la dirección del PAD en la pila.

DUP Duplica el número situado en la cima de la pila que corresponde a la dirección de comienzo del PAD.

1+ Suma uno a la cima de la pila.

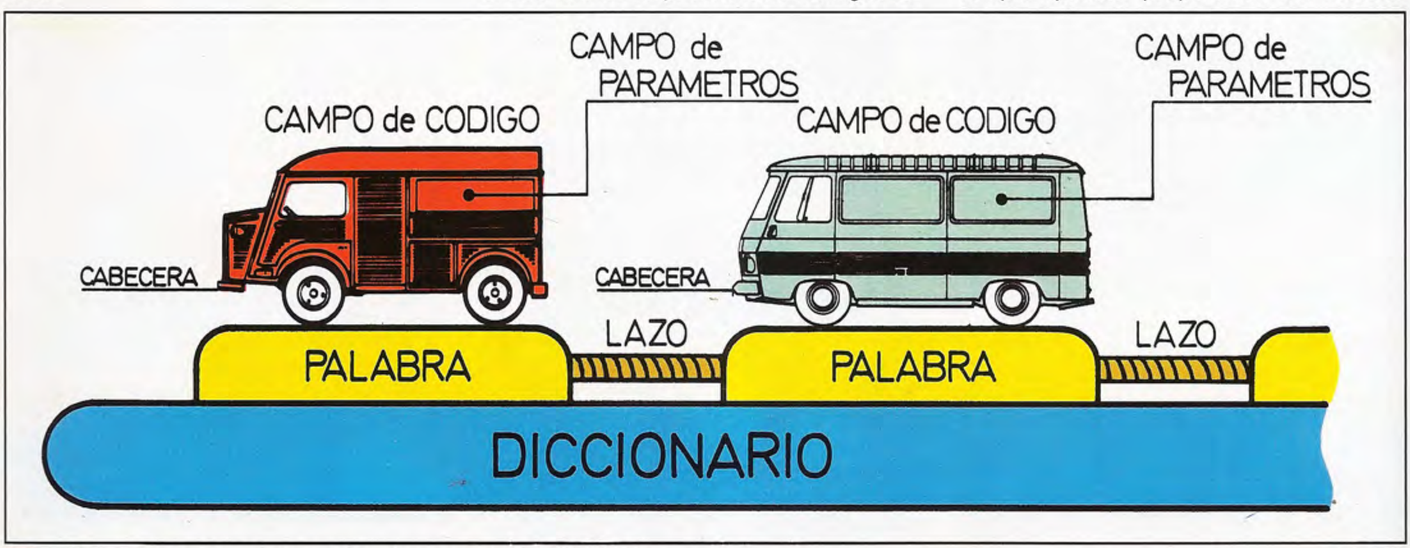
SWAP Intercambia los dos valores de la cima de la pila, con lo cual el elemento más cercano a la cima de la pila es la dirección de comienzo del PAD, mientras que el segundo coincide con esta dirección incrementada en una unidad.

C Lee el primer elemento del PAD (ya que esa es la dirección que se encuentra en la cima de la pila) que coincide con la longitud.

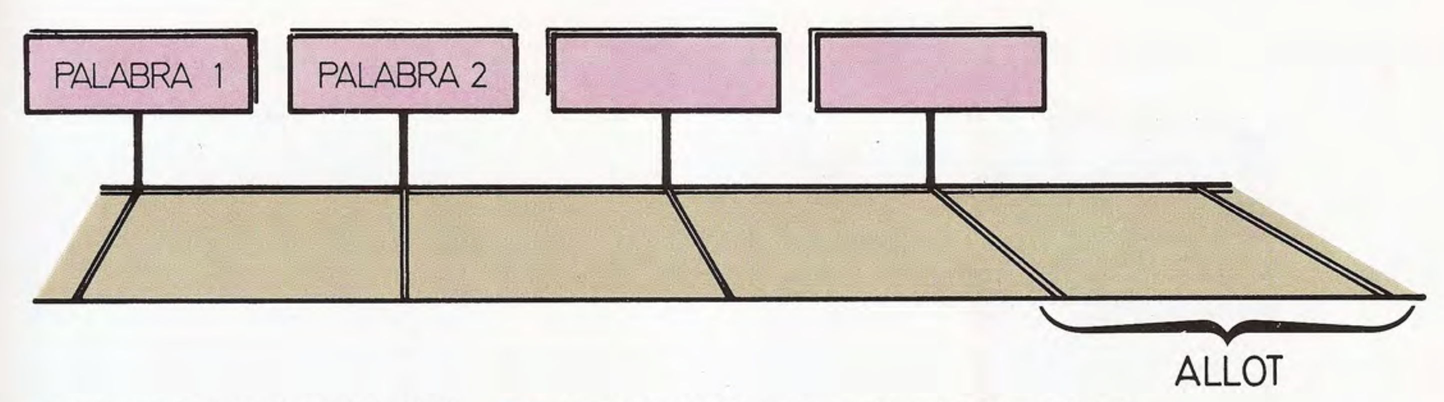
OVER Copia el segundo elemento de la pila y lo deposita en la cima de la misma.

+ Suma los dos primeros elementos de la pila: la longitud de la cadena y la dirección del comienzo del PAD.

SWAP Intercambia los dos elementos más cercanos a la cima de la pila, con lo que quedan preparados los índices



Cada uno de los componentes de una palabra FORTH tiene una misión específica; su síntesis da cuerpo a la palabra en sí.



La palabra ALLOT reserva un determinado espacio para el campo de parámetros de una palabra.

para el array (estos son la dirección final de la cadena en el PAD y la dirección de comienzo).

A partir de aquí aparece un bucle que toma como inicio la dirección donde se encuentra el primer elemento del PAD y como final el último elemento del PAD; extrae uno a uno estos elementos y los transfiere al campo de parámetros de la palabra que se esté definiendo.

Para definir una palabra, como por ejemplo ASURBANIPAL, basta con introducir la siguiente secuencia de palabras:

```
CREATE PALABARA CADENA ASURBANIPAL <CR>
```

La respuesta del ordenador será la siguiente:

```
CREATE PALABA CADENA ASURBANIPAL OK
```

Para leer esta cadena se puede emplear la siguiente secuencia:

```
PALABRA 11 TYPE <CR>
```

Obteniendo como respuesta:

```
PALABRA 11 TYPE ASURBANIPAL OK
```

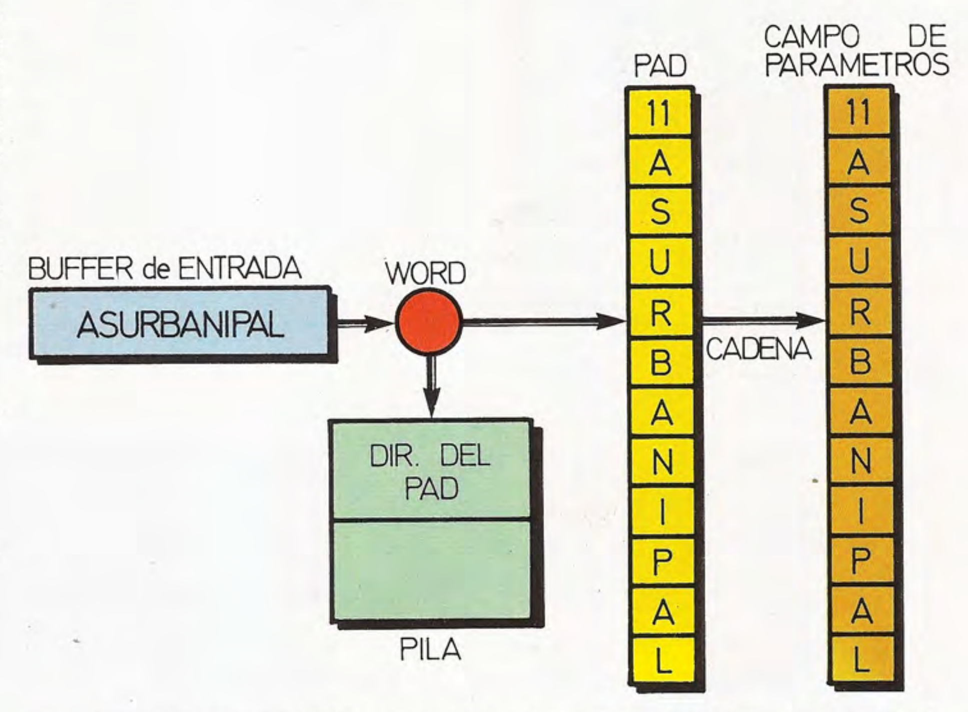
De la misma forma, también es posible introducir un array de caracteres:

```
CREATE SEMANA CADENA LUNMARMIEJUEVIESAB
DOM<CR>
```

La respuesta ahora será la siguiente:

```
CREATE SEMANA CADENA LUNMARMIEJUEVIESABDOM
OK
```

Para visualizar un elemento de este array de caracteres, resulta conveniente definir una nueva palabra que facilite el trabajo:



Mediante la palabra WORD se transfiere el contenido del buffer de entrada al PAD. A su vez, por medio de la palabra CADENA el contenido del PAD pasa al campo de parámetros de una palabra.

```
: INDICE
1 -3 *SEMANA +
3 TYPE
;
```

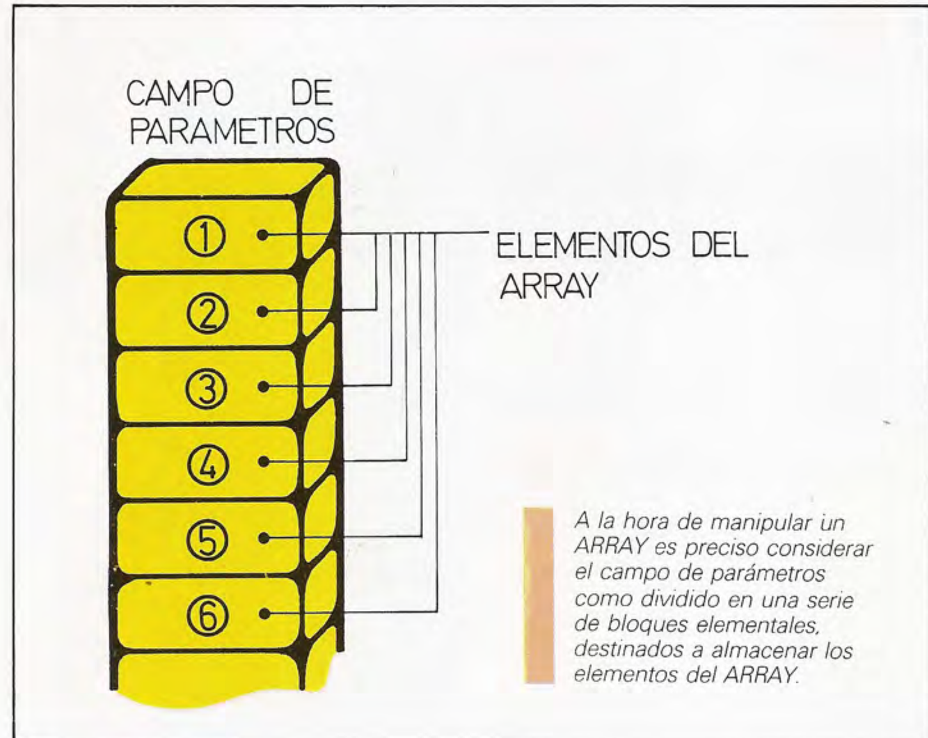
Al ejecutar:

```
3 INDICE <CR>
```

Se obtendría la siguiente respuesta:

```
3 INDICE MIE OK
```

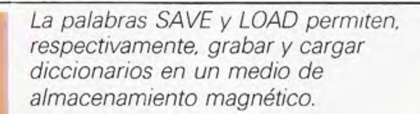
Esta forma de crear palabras presentaron ciertos inconvenientes. Al utilizar la palabra tan sólo pasa a la pila la di-



A la hora de manipular un ARRAY es preciso considerar el campo de parámetros como dividido en una serie de bloques elementales, destinados a almacenar los elementos del ARRAY.

OUT IN ENTORNO

LOAD SAVE



En FORTH, el usuario cuenta con las palabras IN y OUT como herramientas para faciliar la comunicación del ordenador con el exterior.

La palabras SAVE y LOAD permiten, respectivamente, grabar y cargar diccionarios en un medio de almacenamiento magnético.

rección de comienzo de su campo de parámetros, con lo que resulta necesario conocer la longitud de la cadena o la longitud de cada uno de los elementos de la cadena, así como los índices.

Otra palabra que permite definir tipos de palabras es DEFINIR, la cual se emplea ademaś asociada con la palabra DOES>.

Mediante esta definición de tipo de palabras, se especifica en primer lugar cómo se creará esa palabra mediante las palabras emplazadas entre la cabecera y la palabra DOES>; e incluso se puede especificar la acción que se ha de realizar cuando una palabra de este tipo sea invocada. La palabra DOES> crea el campo del código.

```
DEFINIR CADENA
  ASCII "WORD DUP 1+ SWAP C   DP C,
  OVER + SWAP
  DO
    | C  C,
  LOOP
  DOES
    DUO 1+ SWAP C
  ;
```

Por lo que respecta al campo de parámetros, la nueva palabra actúa de la misma forma que la palabra CADENA anteriormente definida; sin embargo, en este caso, por efecto de la definición de un campo de código, al ser invocada la palabra cadena se situará en la pila de la propia longitud de la misma, además de su dirección de comienzo.

A partir de esta forma de almacenamiento de las cadenas se pueden realizar una serie de operaciones con ellas: subdivisión, concatenación, etc.

Manejo de memoria

Las palabras que más directamente manejan la memoria son las siguientes: I y @ que permiten leer o modificar el valor de dos bytes de memoria, y CI y C@ que permiten realizar esa misma operación con un solo byte. Además de estas, existen otras palabras FORTH catalogables en este apartado funcional:

HERE Proporciona una dirección de memoria que corresponde a la primera posición libre tras el diccionario. Esta dirección de memoria se deposiitará en la

cima de la pila a la disposición del operador.



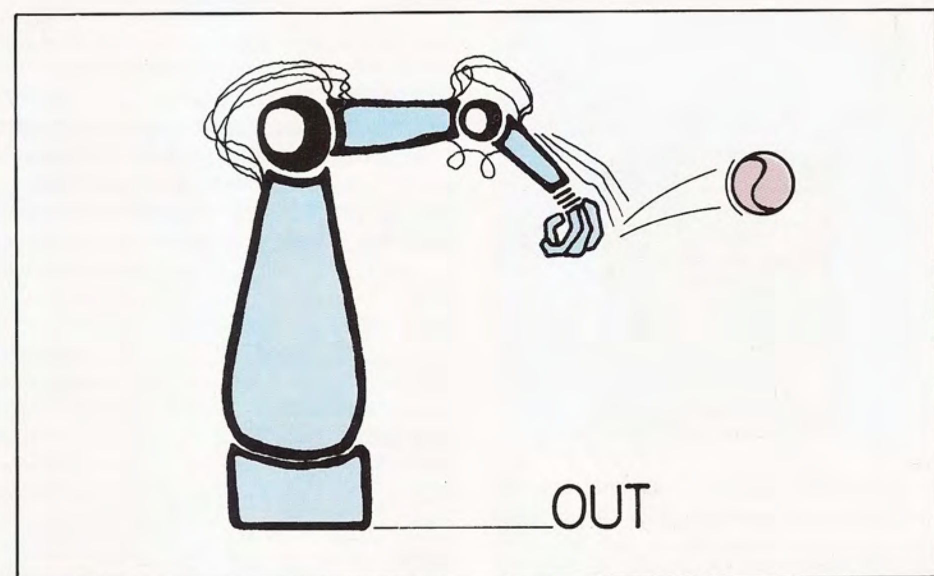
HERE . 15897 OK

CALL Ejecuta una rutina en código máquina situada en una dirección de memoria que se extrae de la pila.

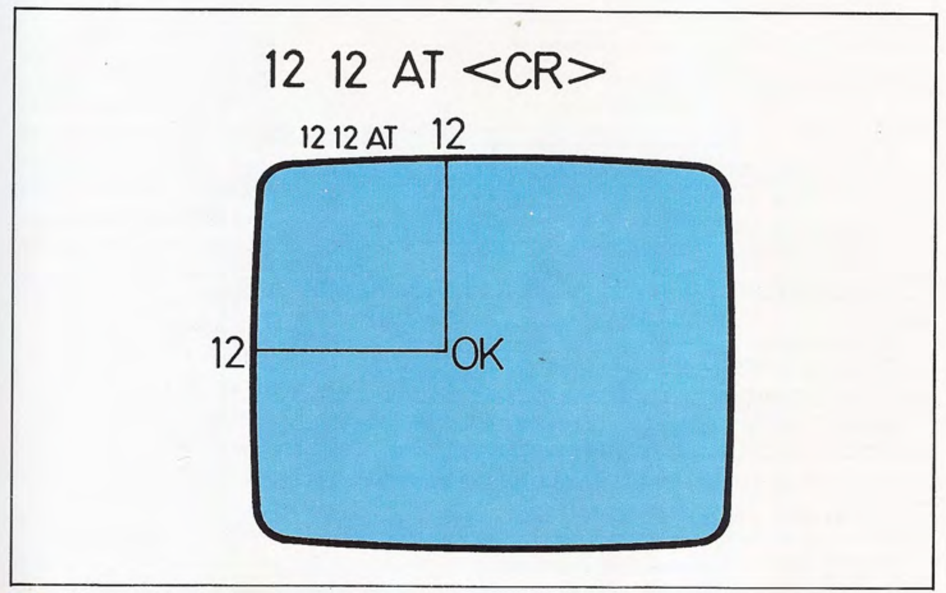
Para manejar las operaciones de Entrada/Salida existen dos instrucciones especializadas:

OUT Envía un dato a un acceso para comunicación externa cuya dirección ha de especificarse.

IN Recoge un dato de un acceso cuya dirección se recoge en la pila.



La palabra OUT manda un byte hacia el exterior o «port» de comunicación.



El cometido de la palabra AT no es otro que posicionar el cursor en un determinado punto de la pantalla.

Inserción de comentarios

Al revisar un programa transcurrido algún tiempo a partir del momento en que fue confeccionado, ocurre con frecuencia que el usuario ha olvidado por completo su estructura y contenido. Para evitar tal inconveniente es preciso incluir algunas notas dentro del programa que, más tarde, permitirán recordar ciertos detalles importantes del mismo. Al efecto, la nota u observación a incluir debe aparecer encerrada entre paréntesis; por ejemplo:

```
: PROGRAMA
2 127 +
(ESTE PROGRAMA SIMPLMENTE SUMA 2 Y 127 E IMPRI
ME EL RESULTADO)
.
;
```

Dentro de las definiciones de palabras

Durante el proceso de definición de una palabra FORTH puede ser deseable pasar a modo intérprete para efectuar alguna operación. A la hora de efectuar estas operaciones hay que recurrir a sendas palabras FORTH coincidentes con los símbolos de «corchete». Veamos un ejemplo:

```
: PRUEBA
[ 2 BASE C! ]
111
[ DECIMAL ]
7 + .
;
```

La anterior definición de la palabra PRUEBA equivale, en definitiva, a la siguiente:

```
: PRUEBA
7 7 + .
;
```

Ya que 111 fue definido a través de un paso intermedio a binario. También es posible definir palabras que actuarán cual si se tratara realmente de una secuencia de instrucciones semejantes a las que se encierran entre las palabras FORTH [ ]. Las referidas palabras se definen según el método habitual y, a continuación, ejecuta la palabra IMMEDIATE:

```
: BASE2
  2 BASE C!
;
IMMEDIATE <CR>
```

Otro ejemplo puede ser:

```
: BASE10
DECIMAL
;
IMMEDIATE<CR>
```

A partir de este momento se puede sustituir la definición realizada anteriormente con la colaboración de las palabras [y] por las palabras que se acaban de definir. Veamos lo que sucedería con la anterior definición:

```
: PRUEBA
BASE2 111
BASE10 7
```

En efecto, su funcionamiento coincidirá con el asociado a la primera definición de las mismas.

LITERAL Es una palabra que puede resultar útil especialmente cuando se trabaja con la palabra [y]. Permiten pasar un elemento de la pila a la definición de una palabra. Veamos un ejemplo:

```
: PRU
[ 2 2 + ] LITERAL
12 +
.
;
```

Al ejecutar esta palabra (PRU LCR>) se obtendrá el siguiente resultado:

```
PRU 16 OK
```

Mientras que si ordenamos un listado de la misma (con LIST PRU <CR>) el resultado en la pantalla será:

Tabla de órdenes FORTH		
PALABRA	DESCRIPCION	TIPO
D+	Permite sumar dos números de doble precisión.	Palabra aritmética
DNEGATE	Cambia el sigo de un número de doble precisión.	Palabra aritmética.
U*	Multiplica dos números enteros sin signo y da como resultado un número de doble precisión.	Palabra aritmética.
D<	Compara dos números de doble precisión y da como resultado un 1 ó un 0 según que uno sea mayor que otro.	Palabra de comparación.
U/MOD	Divide dos números enteros sin signo, proporcionando el resto y el cociente.	Palabra aritmética.
<#	Inicia el paso de dígitos al PAD.	Manejo del PAD.
#<	Termina el paso de dígitos al PAD.	Manejo del PAD.
#	Pasa un dígito al PAD.	Manejo del PAD.
#S	Pasa todos los dígitos que queden al PAD.	Manejo del PAD.
HOLD	Guarda un carácter ASCII en el PAD.	Manejo del PAD.
ASCII	Deja en la pila el código ASCII del carácter siguiente.	Manejo de caracteres.
SIGN	Pasa al PAD el signo de un entero situado en la cima de la pila.	Manejo del PAD.
CREATE	Crea una nueva palabra.	Manejo del diccionario.
,	Transfiere un elemento de la pila al diccionario, reservando para él dos bytes en la zona de parámetros de la palabra.	Manejo del diccionario.
C,	Transfiere un elemento de la pila al diccionario reservando para él un byte en la zona de parámetros de la palabra.	Manejo del diccionario.
ALLOT	Reserva un espacio determinado en la zona de parámetros de una palabra.	Manejo del diccionario.
DEFINER/DOES>	Define un tipo de palabra.	Manejo del diccionario.

Tabla de órdenes FORTH		
PALABRA	DESCRIPCION	TIPO
HERE	Da la dirección del último byte libre tras el diccionario.	Manejo de memoria.
CALL	Ejecuta una rutina en código máquina cuya dirección de comienzo ha de encontrarse previamente en la pila.	Manejo de la memoria.
IN	Recoge un byte de un acceso externo.	Entrada/Salida.
OUT	Envía un byte hacia un acceso de comunicación.	Entrada/Salida.
[	Pasa a modo intérprete.	Palabra auxiliar.
]	Pasa a modo compilación.	Palabra auxiliar.
IMMEDIATE	Hace que la última palabra definida se ejecute dentro de cualquier definición como paso a modo intérprete.	Palabra auxiliar.
(	Comienzo de un comentario.	Inserción de comentarios.
)	Fin de un comentario.	Inserción de comentarios.
LITERAL	Pasa a un valor de la pila a la definición de una palabra.	Palabra auxiliar.
SAVE	Almacena un diccionario en un casete.	Manejo de casete.
LOAD	Recoge un diccionario en un casete.	Manejo de casete.
VERIFY	Comprueba que la grabación de un diccionario se ha efectuado correctamente.	Manejo de casete.
BSAVE	Almacena un bloque de bytes en casete.	Manejo de casete.
BLOAD	Recoge un bloque de bytes del casete.	Manejo de casete.
BVERIFY	Comprueba que un bloque de bytes ha sido grabado correctamente.	Manejo de casete.
AT	Posiciona el cursor en una posición determinada de la pantalla.	Gráficos.
PLOT	Dibuja un pixel en el punto y las coordenadas que se especifiquen.	Gráficos.
INVIS	Hace que no se envíen a la pantalla los mensajes informativos.	Gráficos.
VIS	Hace que vuelvan a aparecer los mensajes informativos.	Gráficos.
BEEP	Emite un sonido de una frecuencia y duración especificada.	Sonido.

```
: PRU
4 12 + .
; OK
```

Manejo del casete

SAVE Permite grabar un diccionario de palabras FORTH en casete (se graban tan sólo las palabras que el usuario ha ido definiendo). Su formato es:

SAVE <nombre>

LOAD Permite leer un diccionario de palabras almacenado en casete:

LOAD <nombre>

VERIFY Permite comprobar si un diccionario grabado en casete es igual que el que se encuentra en memoria.

VERIFY <nombre>

BLOAD Lee un bloque de bytes almacenados en casete.

<comienzo> <longitud> BLOAD <nombre>

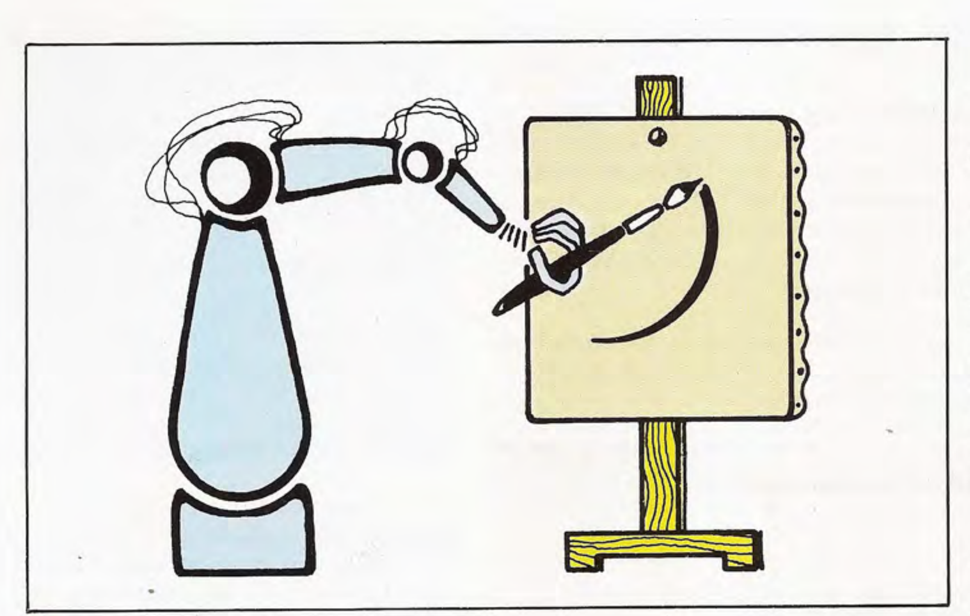
El caso de desear que la dirección de comienzo y la longitud sean las mismas con las que se grabó, sin necesidad de especificar dichos parámetros, hay que utilizar la formulación siguiente:

0 0 BLOAD <nombre>

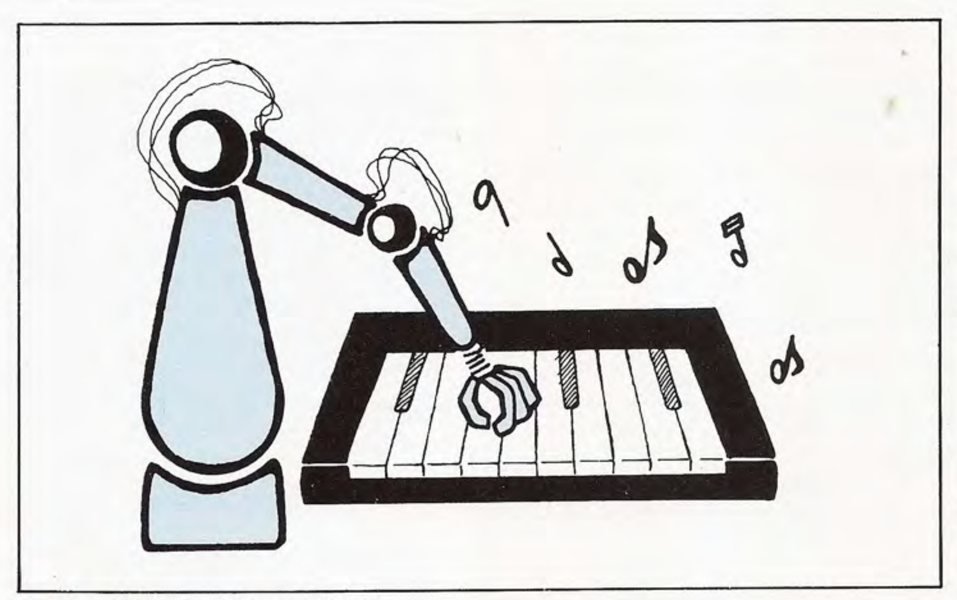
BSAVE Graba un bloque de bytes en casete.

<comienzo> <longitud> BSAVE <nombre>

BVERIFY Compara un bloque de bytes almacenados en casete con el mismo bloque residente en memoria.



El FORTH cuenta con un vocabulario específico para la confección de gráficos.



También es posible en FORTH programar sonidos cuya duración y frecuencia son controlables.

Gráficos en FORTH

AT Permite posicionar el cursor en cualquier punto de la pantalla. Su formulación habitual es:

```
X Y AT
```

PLOT Imprime un pixel en un punto de la pantalla:

```
X Y <modo> PLOT
```

Son cuatro los posibles modos de impresión:

0:Color negro
1:Color blanco
2:No impresión
3:Inversión

Por ejemplo, la ejecución de:

```
12 12 1 PLOT <CR>
```

dará la siguiente respuesta en pantalla:

```
12 12 1 PLOT OK
```

además de imprimir un pixel en la posición de coordenadas 12, 12. Los mensajes que el ordenador visualiza constituyen un inconveniente que es preciso evitar cuando se desea realizar un gráfico en pantalla. Para obviar la aparición de estos mensajes existe una palabra FORTH al efecto:

INVIS Hace que no aparezcan en la pantalla los mensajes habituales.

También existe una palabra FORTH adecuada para ordenar que los mensajes vuelvan a aparecer en el momento oportuno; esta es:

VIS Reaparecen en la pantalla los mensajes generados por la máquina. Es importante conocer el juego de caracteres disponibles en el ordenador que se está utilizando. Al efecto, puede resultar muy conveniente ejecutar la siguiente palabra FORTH que permite visualizar todo el repertorio de caracteres:

```
 CARACT
256 0
DO
  I EMIT
LOOP
```

El sonido

El FORTH también brinda la posibilidad de producir sonidos, a tal efecto existe la palabra BEEP.

BEEP produce dos sonidos necesitando para ello que en la pila se encuentren dos valores. Estos son interpretados como la duración en msg (el valor más cercano a la cima de la pila) y el tono deseado (el siguiente elemento localizado en el pila). El sonido que se produce será agudo cuanto más pequeño sea el número especificado para controlar el tono.

FORTRAN

El precursor de los lenguajes de alto nivel

Si hay que buscar la semilla del árbol genealógico de los lenguajes de programación, cuyas frondosas ramas se extienden por un espacio de tiempo superior a los treinta años, hay que dirigirse en la dirección que señala FORTRAN. Este lenguaje recibe su nombre de la fusión de las iniciales de los vocables ingleses FORmula TRANSlation, que en términos latinos se puede aproximar por «traducción de fórmulas».

El origen del nombre del FORTRAN da la clave para entender la rápida popularización de este lenguaje entre la comunidad científica de los años sesenta. Con anterioridad a su aparición, el personal investigador enfrascado en complicados cálculos tenía que recurrir a igualmente complicados mecanismos de programación para presentar al ordenador, en una sintaxis reconocible para éste, las expresiones matemáticas que tenía que manipular. Con el advenimiento de FORTRAN este tipo de personas encontraron un vehículo que, aunque en la actualidad ya ha sido superado por otros medios de representación, supuso un cambio radical en las relaciones con el ordenador.

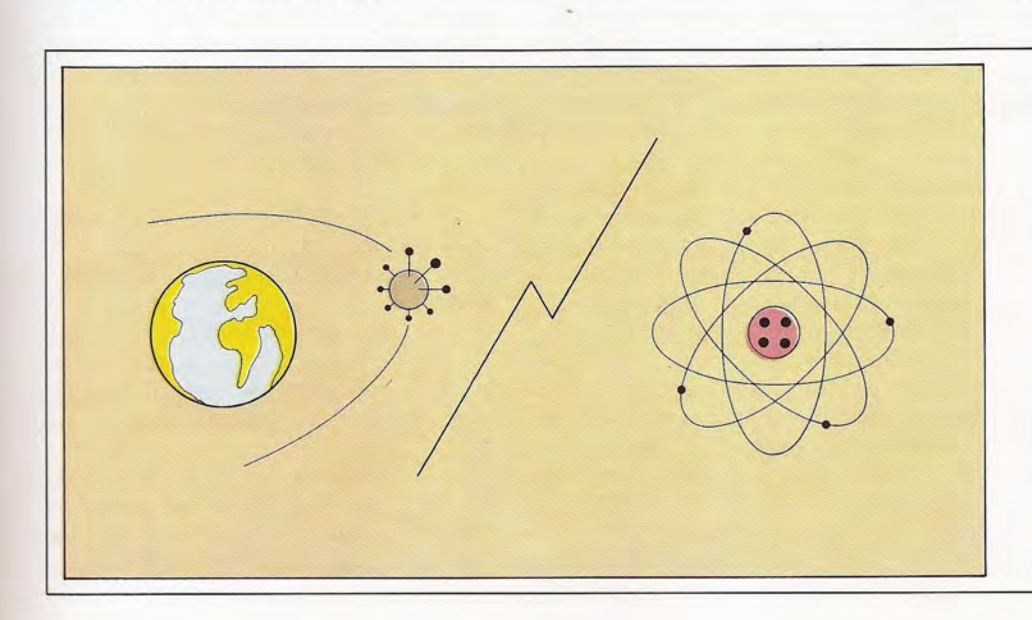


El primer manual de FORTRAN fue publicado por IBM en 1956, año en que comenzaron a redactarse numerosas versiones o dialectos, hasta llegar a nuestros día, en los que el FORTRAN ya ha sido superado en muchos aspectos por otros lenguajes.

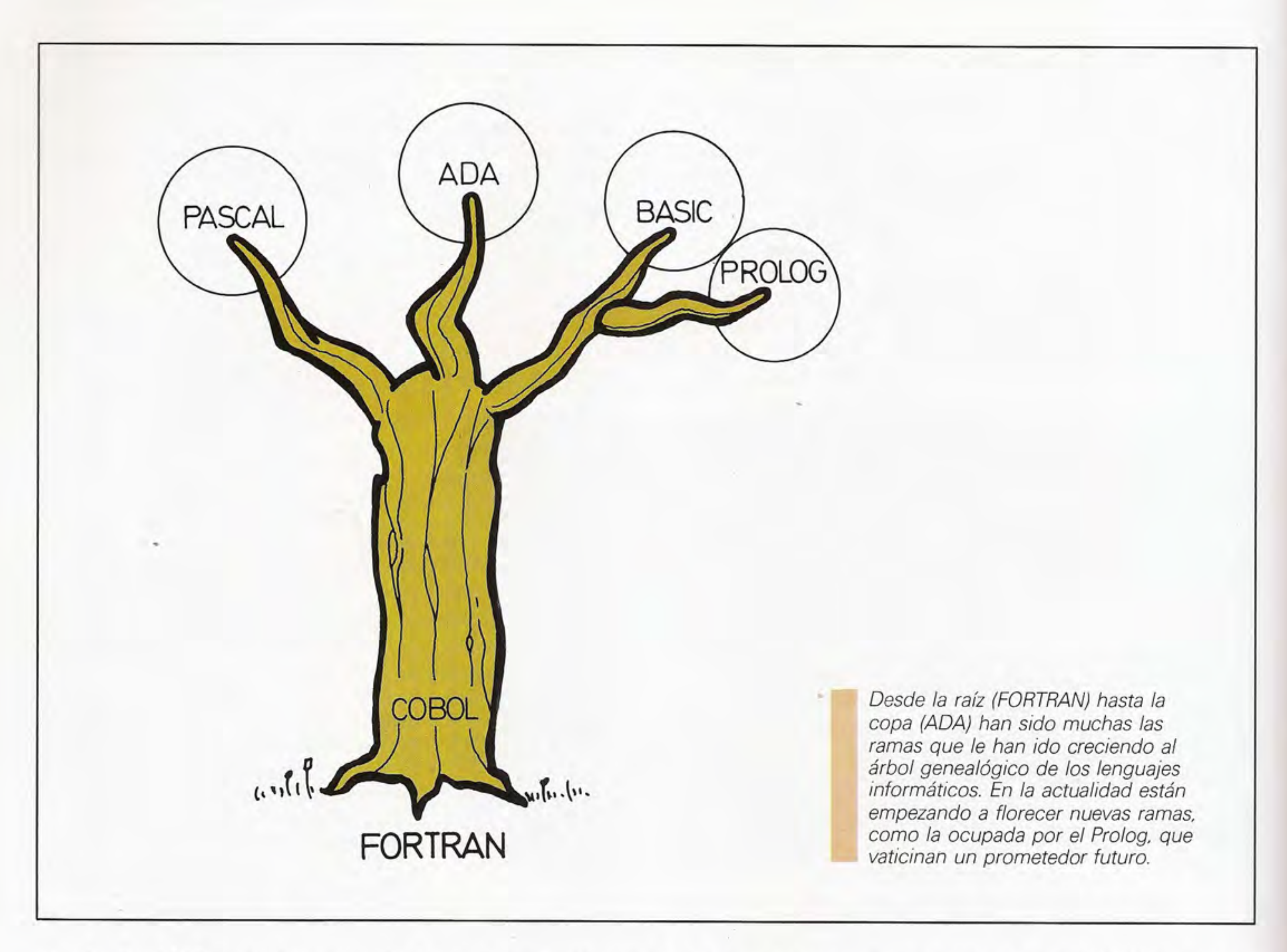
Un poco de historia

FORTRAN es el fruto de la evolución natural de la historia de la informática. El lenguaje ensamblador, que con anterioridad a la aparición de FORTRAN era prácticamente el único medio de comunicación entre el hombre y el ordenador, se encontraba muy cercano a éste. Uno de los objetivos de FORTRAN fue el poner el medio de comunicación más cerca del elemento más importante, el programador, en un esfuerzo por transformar el arte de la programación en la ciencia de la programación. Por otra parte, las características de FORTRAN están muy fuertemente marcadas por la concepción, muy difundida en aquella época y hoy totalmente superada, de que los ordenadores eran fundamentalmente herramientas científicas cuya principal misión era la de realizar cálculos científicos. Muy pronto, la aparición del Cobol hizo que esta suposición empezara a tambalearse, cayendo definitivamente con lenguajes como Lisp o, más recientemente, Prolog.

La idea de que algo parecido al FORTRAN era necesario ya estaba presente



El lenguaje FORTRAN está orientado especialmente a aplicaciones de tipo científico y técnico, desde la determinación de las órbitas de satélites espaciales hasta la definición de la distancia existente entre átomos.



Desde la raíz (FORTRAN) hasta la copa (ADA) han sido muchas las ramas que le han ido creciendo al árbol genealógico de los lenguajes informáticos. En la actualidad están empezando a florecer nuevas ramas, como la ocupada por el Prolog, que vaticinan un prometedor futuro.

en el año 1949, cuando apareció el SHORT-CODE creado por el Dr. Mandy para Univac. Cuatro años más tarde (en 1953), y con la misma idea, apareció el SPEED-CODING de Backus en 1953, esta vez para IBM.

Fue el propio Backus quien en el año 1954 comenzó el desarrollo del lenguaje FORTRAN, también en el seno del gigante americano, viendo la luz el primer manual del lenguaje en el año 1956. En los años en los que se centra nuestra historia no estaba todavía desarrollada la llamada «Teoría Formal de Lenguajes» (ver cuadro de texto al respecto) la cual constituye uno de los mejores ejemplos de la interacción de los campos teóricos y prácticos de una ciencia. Con esta teoría, un reducido grupo de programadores pueden desarrollar un compilador libre de errores en un 99% de su código en unas pocas semanas, lo cual contrasta con los años y escasos resultados que consiguieron Backus y sus colaboradores en el desarrollo del primer compilador para este lenguaje.

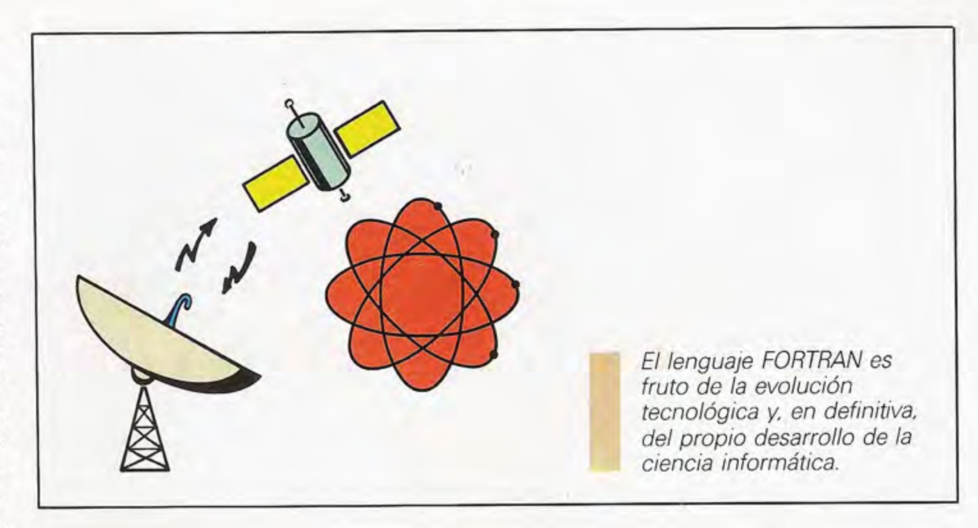
Con anterioridad al desarrollo de la citada teoría, la construcción de un compilador era una tarea que seguía un proceso inconcebible en una ciencia que tradicionalmente se considera como máxima exponente de la rigurosidad: El lanzamiento de la primera versión comercial estaba seguido por una avalancha de protestas de los usuarios que detectaba errores achacables al compilador al desarrollar sus programas. El equipo de programadores de la empresa constructora aislaba la causa del fallo y, si era posible, se daban instrucciones a la empresa que lo detectó para que «parcheara» el programa original de forma que se soslayara el error. Otras veces la solución no era tan sencilla, pues se requería un replanteamiento de buena parte del diseño del producto. Una vez que se habían acumulado un cierto número de «grandes errores» y que se habían mejorado algunos aspectos del lenguaje, llegaba el momento para el lanzamiento de una nueva versión.

De esta forma se llegó al año 1962 con la cuarta versión del lenguaje, la cual recibía el evidente nombre de FORTRAN IV. A partir de aquí la Teoría Formal de Lenguajes empezó a dar sus frutos y la experiencia ganada en el desarrollo de compiladores hicieron que se abandonara esta forma caótica de trabajar.

En la actualidad es posible encontrar un buen número de compiladores de FORTRAN, los más populares de los cuales son el FORTAN-77 y el FORTRAN-80, para ordenadores personales.

Como en las primeras versiones del lenguaje, son los programadores de aplicaciones numéricas y científicas los principales usuarios de estos compiladores. Otras razones para que el FORTRAN pese a lo anticuado que resultan sus estructuras de control y sus primitivas o nulas estructuras de datos, siga siendo utilizado con profusión, son la existencia de un gran número de subrutinas ya desarrolladas y listas para ser utilizadas sin más conocimiento que su nombre, y la gran eficiencia en cuanto a velocidad de ejecución del código generado por un compilador de este lenguaje. Sobre este último punto, es bastante normal encontrar aplicaciones que han sido desarrolladas en un lenguaje cuya estructura se adecúa más a la aplicación concreta y, una vez que se comprueba el buen funcionamiento de las mismas, reprogramar las rutinas aritméticas en FORTRAN para dar una mayor velocidad al producto final.

Como se podrá comprobar a lo largo de esta introducción al FORTRAN, son muchas las similitudes que se encuentran entre este lenguaje y el conocido BASIC. De hecho, el segundo nació como una revisión del primero y, contrariamente a su uso mayoritario en la actualidad, se comenzó a emplear como lenguaje científico en algunas universidades norteamericanas, acabando en lo que hoy todos conocemos.



El lenguaje FORTRAN es fruto de la evolución tecnológica y, en definitiva, del propio desarrollo de la ciencia informática.

Estructura de un programa

En realidad es difícil, si no imposible, hablar de la estructura de un programa FORTRAN de la misma forma en la que se habla de la estructura de un programa C o en Pascal. Las ideas sobre estructuras de datos y modularización de programas que están omnipresentes en estos lenguajes no tienen reflejo en FORTRAN, por el simple hecho de que en los años sesenta este tipo de cuestiones apenas si eran comentario de un grupo reducido de investigadores.

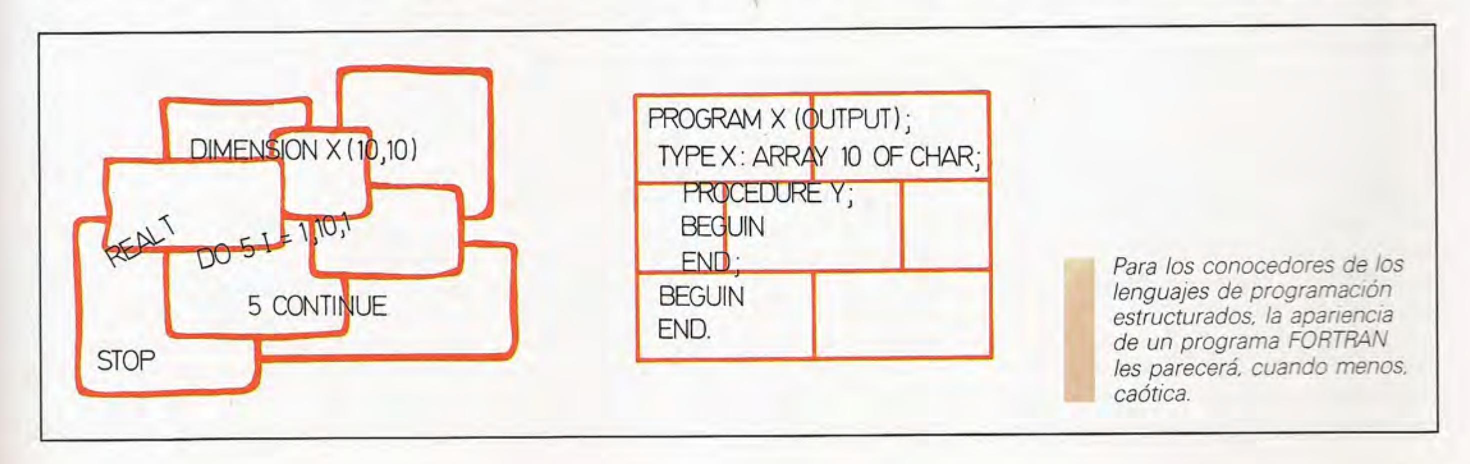
La única estructura de datos avanzada que se encuentra en FORTRAN es el «array» o matriz, y la modularización de los programas sólo puede realizarse a través de las FUNCTIONs y SUBROUTINEs, que serán comentadas posteriormente, y que apenas son un pálido reflejo de las funciones y procedimientos de Pascal.

En FORTRAN no existe una zona de declaración de variables propiamente dicha, ya que algunas de ellas tienen un tipo implícito asociado en función de la primera letra del nombre (hablaremos más despacio de ello), aunque esta declaración puede sustituirse por una explícita realizada por el programador. Tampoco existen la estructuración de tipos como ocurre en Pascal, Ada y Modula-2, o en menor grado en C, ni estructuras de control avanzadas al estilo de las WHILE-DO o REPEAT-UNTIL.

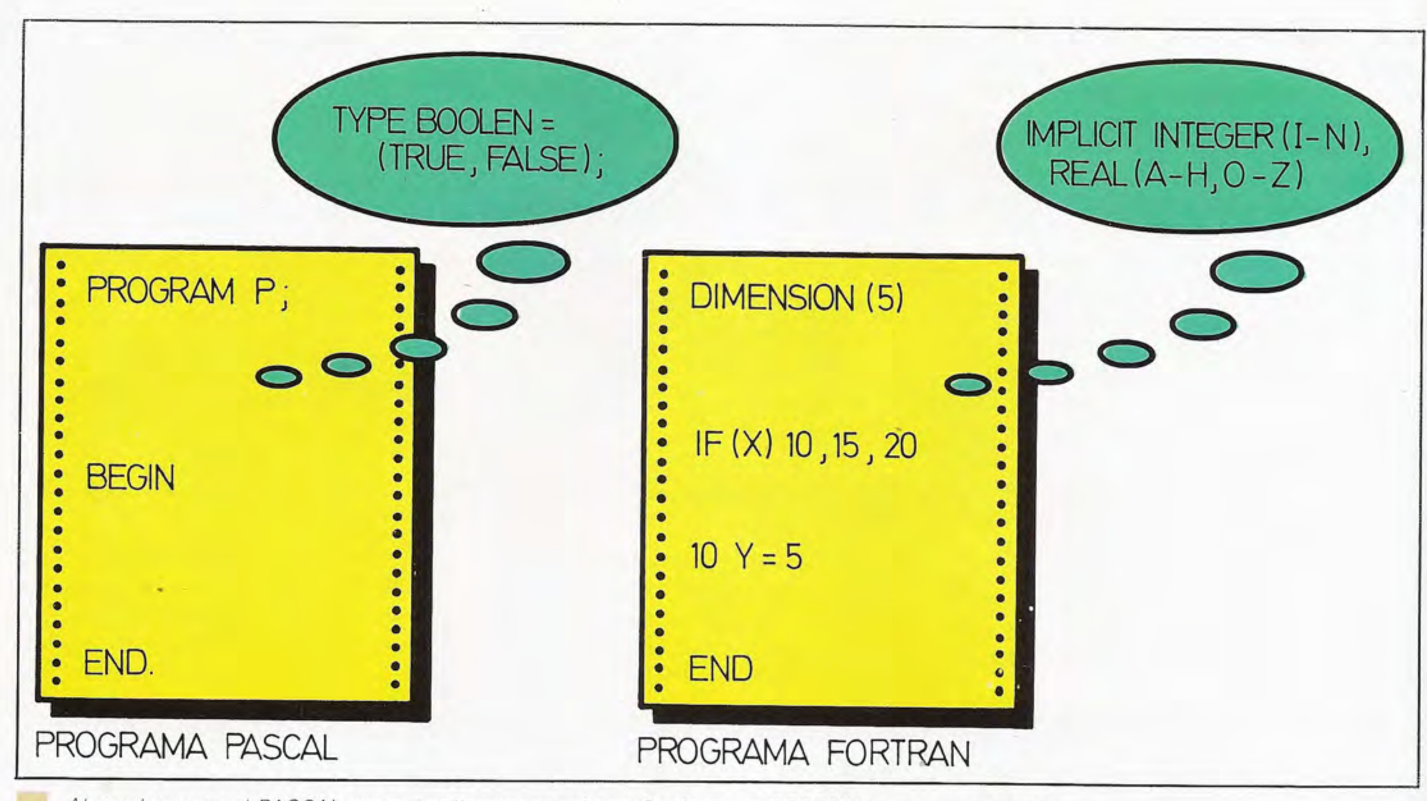
Como ya se ha comentado, para enfrentarse al FORTRAN hay que olvidarse en cierta forma de los postulados habitualmente encontrados en los lenguajes de tipo estructurado y tener en la mente las estructuras características del lenguaje BASIC, mucho más acordes con lo que se encuentra en el protagonista de estos párrafos.

Variables FORTRAN

Una variable en FORTRAN es una cadena de uno a seis caracteres alfanuméricos de los que el primero ha de ser obligatoriamente una letra. La declara-



Para los conocedores de los lenguajes de programación estructurados, la apariencia de un programa FORTRAN les parecerá, cuando menos, caótica.



Al igual que en el PASCAL existe implícitamente el tipo Boolean, en FORTRAN son consideradas enteras ciertas variables y reales las restantes, a no ser que se especifique lo contrario.

ción de variables enteras y reales se puede hacer implícita o explícitamente, esta última a través de las sentencias de «declaración explícita». Una variable que no ha sido declarada explícitamente será considerada implícitamente en FORTRAN como entera siempre que comience por I,J,K,L,M o N, y real si tal variable comienza por una letra distinta de las anteriores (A...H u O...Z).

Los distintos tipos de variables que pueden aparecer en un programa FORTRAN son los siguientes:

* *INTEGER*

Equivalentes a sus homónimos de Pascal. En un programa en el que no haya sentencias de declaración explícita, variables como I, N o NUMERO serán consideradas como enteras. Obsérvese lo acertado de la norma de declaración implícita, según la cual las variables habitualmente empleadas como subíndices en expresiones matemáticas (y que, por tanto, son números enteros) como I o N no necesitan de ninguna declaración especial.

* *REAL*

Representan variables en punto o coma flotante. Variables como ENTERA o DISCRIMINANTE serán consideradas como reales siguiendo la norma de la declaración implícita.

* *DOUBLE PRECISION*

Son variables que, teóricamente, tienen el doble de precisión que las reales. Son equivalentes al tipo «double» del lenguaje C. Su declaración se realizará obligatoriamente a través de una sentencia de declaración explícita.

Mientras que una variable del tipo REAL puede representarse indistintamente en notación de punto flotante o exponencial, las de DOUBLE PRECISION sólo admiten la notación exponencial, con la particularidad de que la letra E que precede al exponente es sustituída por la D. Así, 2.198E+03 es una constante real (de simple precisión), mientras que 2.198D+03 es una constante del tipo «double precision».

* *LOGICAL*

Son equivalentes al tipo Boolean de lenguajes como Pascal, Ada o Modula-2. En BASIC no existen variables con funciones similares. Sólo podrán tomar los valores .TRUE. o .FALSE. (¡Atención a los puntos!) y su declaración, como en el tipo anterior, ha de hacerse obligatoriamente por medio de una sentencia de declaración explícita.

* *COMPLEX*

La frecuente aparición de números complejos en los cálculos matemáticos y la orientación de FORTRAN hacia tareas científicas motivaron la aparición de este tipo de variables, cuya misión es la de soportar números complejos. Tanto la parte real como la imaginaria se consideran números reales. Un ejemplo

de constante de este tipo podría ser (3.5,4.3) que representa el complejo de parte real 3.5 e imaginaria 4.3.

Una sentencia de declaración explícita tiene el siguiente aspecto:

<tipo> <lista de variables>

donde tipo es una de las cinco palabras clave vistas más arriba. Ejemplos de declaraciones explícitas podrían ser:

```
INTEGER VALOR
REAL I1,ABC
LOGICAL ESTADO, FIN
```

Vemos que I1 ha sido declarada como REAL. Una declaración explícita como la que se ha hecho para esta variable rompe con la norma que implícitamente consideraría a I1 como variable entera por empezar por la letra I. Sin embargo dicha norma sigue siendo válida para otras variables que empiecen por tal letra.

Este tipo de sentencias han de ir al principio del programa y se ha de verificar que el mismo nombre de variable no aparezca en más de una sentencia de declaración.

Siguiendo con el tema de las declaraciones explícitas, la sentencia IMPLICIT tiene por objeto declarar de un tipo determinado todas aquellas variables que empiecen por un carácter alfabético dado. Su misión es, por tanto, muy semejante a la de las sentencias DEFINT o DEFSTR de algunos dialectos del BASIC. Un ejemplo de declaración implícita puede ser:

```
IMPLICIT DOUBLE PRECISION (A,D-X)
```

lo que hará que todas las variables que empiecen por A o por D,E,...,V,W,X sean consideradas como de doble precisión.

De igual manera podemos decir:

```
IMPLICIT INTEGER (I-N), REAL (A-H,O-Z)
```

aunque ello no es necesario, ya que se trata precisamente de la declaración implícita de variables reales y enteras que por defecto asume el FORTRAN.

La única estructura de datos avanzada que ofrece el FORTRAN es el array, con la particularidad de que los índices de cualquier elemento referenciado han



Actualmente existen versiones de FORTRAN (como el FORTRAN-77 o el FORTRAN-80) desarrolladas específicamente para ordenadores personales.

de ser siempre mayores que cero. La declaración de un array se hace normalmente por medio de la sentencia DIMENSION. Por ejemplo, las sentencias:

```
DIMENSION MATRIZ (10,10)
REAL MATRIZ
```

declaran la matriz MATRIZ de 10×10 elementos reales. También existe la posibilidad de prescindir de la palabra clave DIMENSION, como en el siguiente ejemplo, por medio del cual se obtiene el mismo resultado que con las dos líneas anteriores:

```
REAL MATRIZ (10,10)
```

Estructuras de control

La tabla que acompaña al texto contiene un resumen de tales estructuras, las cuales serán tratadas con mayor detalle a continuación.

* GO TO incondicional. Análogo al GOTO de Basic. Su forma es:

```
GO TO n
```

En FORTRAN no es necesario preceder a cada línea del programa con un número como ocurre en el lenguaje BASIC. Sólo habrá que incluir los números de línea en aquellas sentencias que, bien sea por ser el destino de un salto o el final de un bucle, tienen que ser referenciadas en alguna otra sentencia.

La «n» de la instrucción GO TO es un número entero que corresponderá a uno de tales números de sentencia.

* GO TO calculado. Su forma es como sigue:

```
GO TO (n1,n2,...,nk),i
```

en donde las «n» son números de sentencia y la «i» una variable entera cualquiera con la condición de que no sea miembro de un array. Cuando el flujo de programa llegue a este punto, se producirá el salto a la sentencia nj si «i» vale j. Si «i» es menor que 1 o mayor que K, la sentencia ejecutada será la que se encuentre justo a continuación del GO TO.

* GO TO asignado. Su formulación es:

```
GO TO i,(n1,n2,...,nk)
```

El tipo y significado de las ni es análogo al caso anterior. La diferencia es que la variable «i» tomará los valores de n1,n2,...,nk en lugar de 1,2,...k. Además, el valor concreto que en cada momento deba tomar la variable «i» ha de ser asignado obligatoriamente a través de una sentencia especial, la ASSIGN, de esta forma:

```
ASSIGN j TO i
```

en donde «i» es la variable del GO TO y «j» una constante entera cuyo valor es el que ha de tomar dicha variable.

* IF aritmético. Su aspecto es:

```
IF (a) n1,n2,n3
```

donde «a» es una expresión matemática no compleja y n1,n2,n3 son números de sentencia. La bifurcación se realizará hacia una de estas dependiendo de que el valor resultante de evaluar «a» sea negativo, nulo o positivo respectivamente.

* IF lógico. Adopta la siguiente formulación:

```
IF (a) b
```

donde «a» es una expresión lógica cualquiera y «b» una sentencia ejecutable (es decir, no de tipo declarativo como IMPLICIT, DIMENSION, etc) excepto una DO u otra IF lógico.

En la correspondiente tabla se resumen los operadores que pueden formar parte de una expresión lógica cualquiera. Como es habitual en todos los lenguajes de programación, nos encontramos con las funciones booleanas habituales. Un detalle de sintaxis muy importante, cuyo olvido trae como consecuencia un buen número de errores, es la presencia de los dos puntos que delimitan cualquier operador lógico.

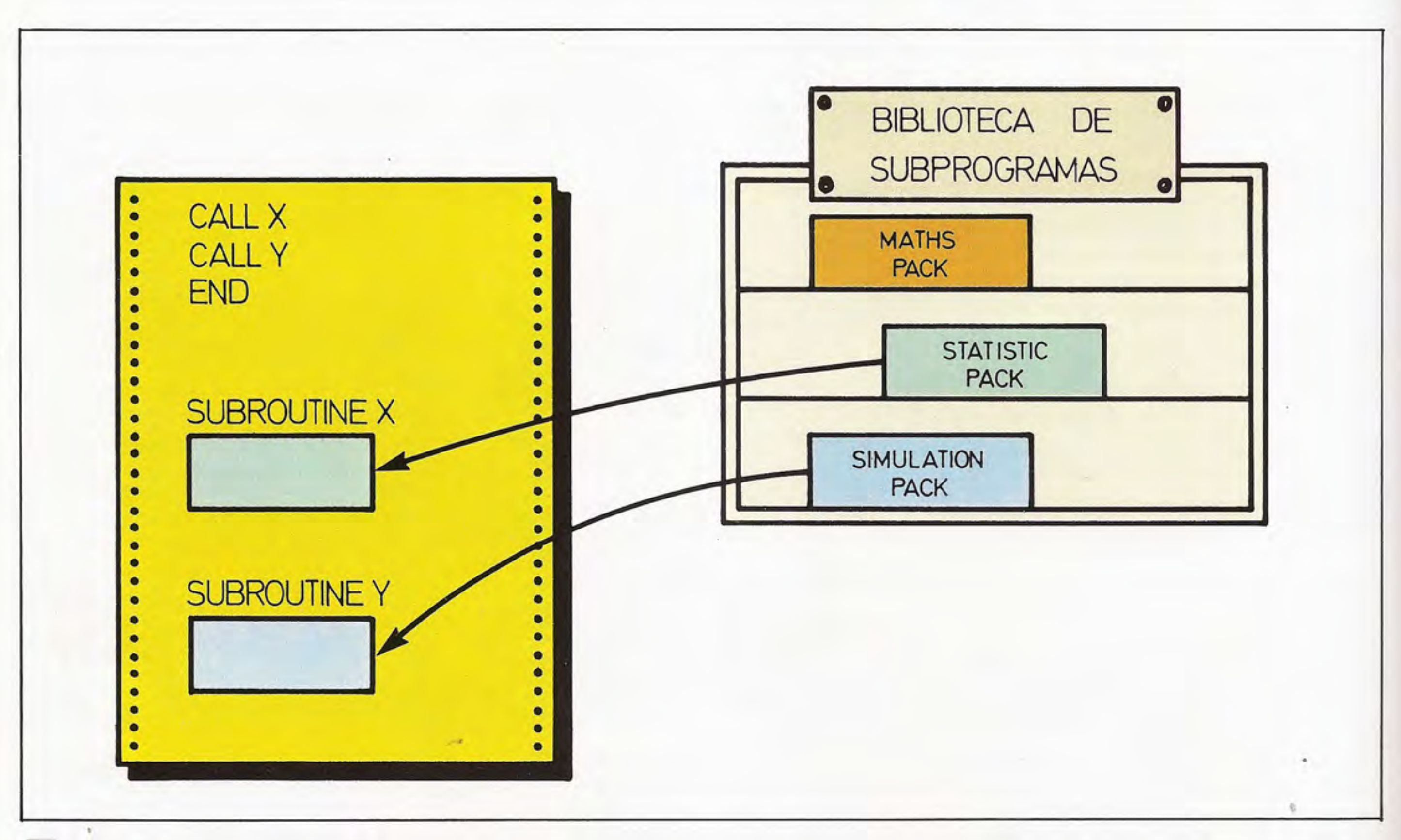
* DO. La misión del DO de FORTRAN es análoga a la de la combinación FOR/NEXT del BASIC: construcción de bucles. Su forma toma el siguiente aspecto general

```
DO n i=m1,m2,m3
```

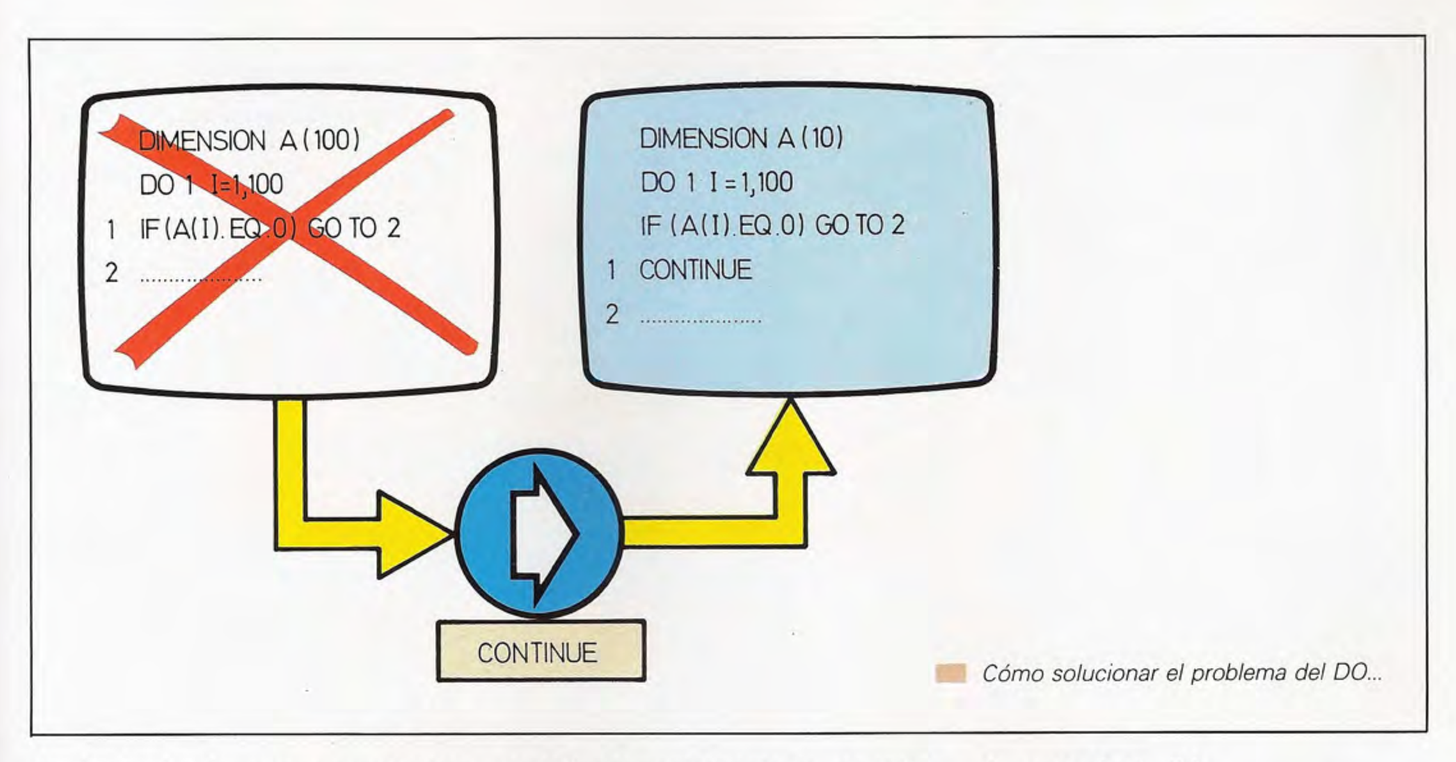
en donde «n» es el número de una sentencia ejecutable que aparecerá en el listado con posterioridad a la propia DO; puede ser cualquiera excepto otra DO, GO TO, IF, STOP o RETURN. «i» es una variable entera no perteneciente a un array y m1,m2,m3 son constantes o variables enteras; en el caso de que se trate de variables, tampoco pueden pertenecer a un array.

En operación, el rango de sentencias desde la propia DO hasta la «n» se ejecuta para valores de «i» desde m1 a m2, con incrementos de valor m3. Este último parámetro es opcional y su valor por defecto es uno.

Si la última sentencia del bucle ha de ser una de las prohibidas, se soluciona la situación con haciendo que sea una sentencia «CONTINUE» la que marque el final del bucle. El único efecto de la sentencia CONTINUE es devolver el control a la cabeza del bucle para iniciar otra ecuación. No es infrecuente el que to-



La union hace la fuerza...



Cómo solucionar el problema del DO...

dos los bucles de un programa Fortran tengan como última sentencia una CONTINUE.

* PAUSE. Al aparecer esta sentencia se detiene momentáneamente el programa, dando opción al usuario a reinicializarlo o detenerlo definitivamente.

* STOP. Esta sentencia marca el final lógico de un programa.

* END. Indica el final físico del programa.

En FORTRAN se hace una distinción clara entre final lógico y final físico de un programa. Los que hayan experimentado con el BASIC saben que la última sentencia que se ejecuta en un programa no tiene por qué ser siempre precisamente la última sentencia que aparece en el listado. Sentencias BASIC como «stop» o «end» pueden detener la ejecución del programa en cualquier punto aunque queden todavía sentencias por ejecutar en el curso normal del mismo.

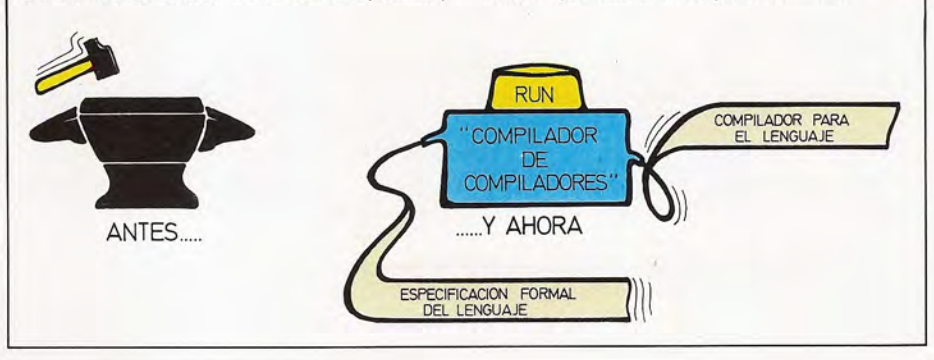
La forma habitual de programar en FORTRAN consiste en disponer los subprogramas al final del código del programa principal. En algún punto del programa principal debe existir una sentencia STOP que marque el final de la ejecución. La sentencia END es normalmente la última de cualquier programa, y se trata fundamentalmente de una indicación al compilador para que deje de traducir código fuente. Su presencia es siempre obligatoria.

La entrada/salida

La entrada/salida es uno de los aspectos más primitivos y tediosos del

La teoría formal de lenguajes

Los primeros compiladores para lenguajes de alto nivel constituían un buen ejemplo de labor puramente artesanal. Al no existir un fundamento teórico sobre la construcción de tales programas, su confección se basaba en recopilar las experiencias anteriores y en pruebas de ensayo y de error.
Por las fechas en las que el FORTRAN IV vio la luz del día, una serie de lingüistas, fundamentalmente Noam Chomsky y Ferdinand de Saussure habían desarrollado unas teorías sobre el lenguaje natural que constituyeron la «gramática estructuralista». Sus principales pilares son la división de las oraciones en estructura profunda y superficial, y la división de las mismas en sintagmas. Estos lingüistas, en colaboración con expertos en ordenadores, aplicaron los resultados de la gramática estructuralista al campo de los lenguajes de programación, sentando las bases de la Teoría formal de lenguajes.
La consideración de esta teoría a la hora de diseñar un nuevo lenguaje supone disponer de un compilador para el mismo en un tiempo record, frente a lo que se tardaría en el caso de no hacerlo así.
La herramienta más sorprendente, fruto de la aplicación de esta teoría, es un programa llamado «compilador de compiladores»; el cual, a partir de una especificación de un lenguaje, es capaz de realizar gran parte del esfuerzo necesario para diseñar un compilador del mismo.



```
      INTEGER FUNCTION FACT (*, X)
      INTEGER X
      FACT=1
      IF (X-1) 1, 4, 2
   1  RETURN 1
   2  DO 3  I=2, X
   3  FACT=FACT*I
   4  RETURN
      END
```

Ejemplo de llamada:

```
      I=FACT($10,5)
```

PROGRAMA 1. Función FORTRAN para calcular el factorial de un número.

FORTRAN. Este descuido se debe en gran medida a la orientación del lenguaje hacia tareas científicas, en las que el volumen de cálculo es muy superior al de entrada/salida. Lo normal en un programa escrito en FORTRAN es generar un número muy grande de operaciones matemáticas a partir de un conjunto reducido de datos o que, en el caso de que sean numerosos, se repiten según patrones bien definidos. Igual comentario se puede realizar sobre la salida, la cual está constituida por unos pocos valores o por un gran número de ellos pero presentables según el mismo formato.

La estructura general de una sentencia de entrada/salida en FORTRAN toma el siguiente aspecto:

```
<código de operación> (<número de fichero>,<formato>)
<lista de variables>
```

El «código de operación» puede ser una de las palabras clave READ o WRITE, según se trate de una lectura o escritura respectivamente, y la «lista de variables» son las variables cuyos contenidos se van a leer o escribir. El «número de fichero» especifica el dispositivo de E/S a utilizar: pantalla, teclado, impresora, lectora de tarjetas perforadas (... en los viejos tiempos), etc. Este número depende de la configuración del ordenador utilizado, y varía de unos equipos a otros. En el campo <formato> se incluye un número entero que será el de una sentencia FORMAT, en la cual se especifica la forma en la que se ha de realizar la entrada/salida, de forma muy parecida a como lo hace el string de formato de las sentencias «printf» o «scanf» del lenguaje C. Por ejemplo, el siguiente segmento de programa:

```
      INTEGER ENTERO
      ENTERO=1
      REAL=3.14
      WRITE (5,500) ENTERO,REAL
500   FORMAT (I5,F10.2)
      END
```

hará que se escriban las variables ENTERO y REAL a través del dispositivo que tiene asociado el número 5 (la pantalla, por ejemplo), en el formato que se

```
      SOBROUTINE FACT(*, X, XFACT)
      INTEGER X, XFACT
      XFACT=1
      IF (X-1) 1, 4, 2
   1  RETURN 1
   2  DO 3  I=2, X
   3  XFACT=XFACT*I
   4  RETURN
      END
```

Ejemplo de llamada:

```
      CALL FACT ($10,VALOR,RESUL)
```

PROGRAMA 2. Subrutina para el cálculo del factorial de un número. En el ejemplo de llamada, «VALOR» y «RESUL» han sido declaradas previamente como variables enteras.

especifica en la sentencia FORMAT cuyo número es el 500. El formato «I5» reserva un campo de 5 posiciones que albergarán a un entero, mientras que el «F10.2» reserva diez posiciones en total para escribir un número real en formato de punto flotante con dos de ellas dedicadas a la parte decimal. El resultado de la ejecución de este programa sería por tanto:

```
bbbb1bbbbb10.2b
```

en donde los caracteres «b» significan espacios en blanco.

Especificadores de campo y caracteres de control

/	Salto de línea.
Wx	Deja «x» espacios en blanco a partir de la posición en que estaba.
Tx	Coloca la cabeza en la posición «x».
Ix	Formato para campo entero de «x» posiciones.
Fx.y	Formato para campo real en punto flotante de «x» posiciones, estando «y» de ellas reservadas a la parte decimal.
Ex.y	Formato para campo real en notación exponencial de «x» posiciones, estando «y» de ellas reservadas al exponente.
Dx.y	Como Ex.y aunque para variables de doble precisión.
Lx	Especifica un campo de «x» posiciones para una variable lógica.
''	Los caracteres que estén escritos entre comillas simples serán escritos directamente.



El mismo comentario que se hizo sobre «scanf» en el lenguaje C es aplicable a la sentencia READ: el formato de la misma actúa como un filtro de la entrada que se está tecleando, admitiendo tan sólo los datos que estén de acuerdo con el mismo.

En la sentencia FORMAT, al igual que en el string de formato de C, se pueden incluir caracteres de control cuyo propósito es el salto de líneas o páginas en el dispositivo escritor o realizar borrados de pantalla en el caso de que este se encuentre materializado en un terminal de video. La mayor parte de ellos aparecen en la correspondiente tabla junto con los principales especificadores de campo.

La sentencia FORMAT, al igual que otras como IMPLICIT o DIMENSION, pertenecen al conjunto de las «no ejecutables», nombre que reciben por el hecho de que el compilador no genera código para ellas.

Los subprogramas: FUNCTIONs y SUBROUTINEs

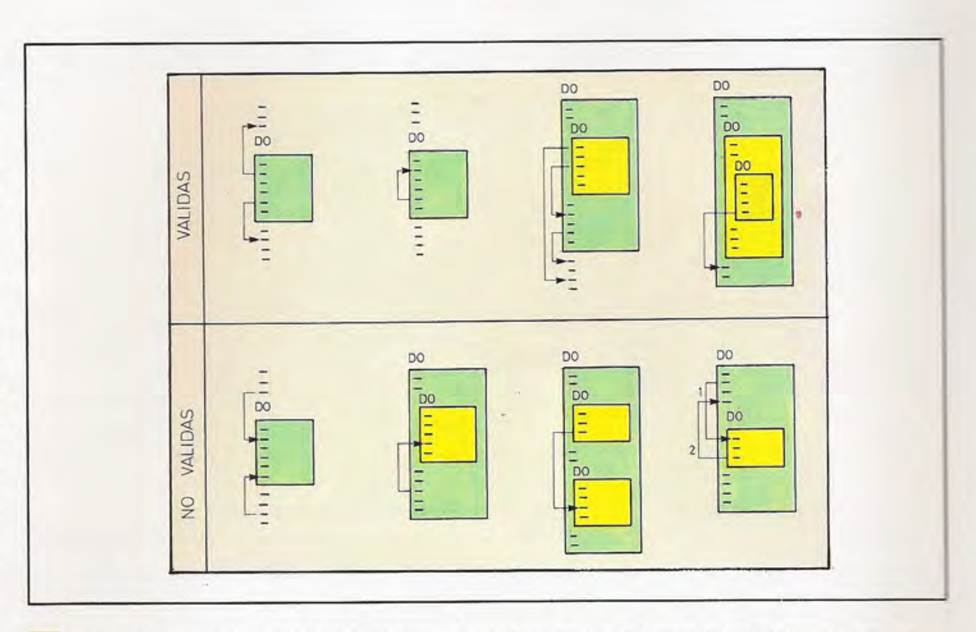
La declaración de una FUNCTION se realiza de la siguiente forma:

<tipo> FUNCTION nombre (a1,...,an)

El concepto de FUNCTION en FORTRAN es idéntico al que se encuentra en otros lenguajes de programación: se trata de conseguir un resultado a partir de unos parámetros. En FORTRAN el tipo de resultado devuelto está determinado por el nombre de la función según la regla de los tipos implícitos en el caso de que no aparezca «<tipo>». Si no ocurre así, es esta partícula la que determina el tipo de valor resultado de operar con los parámetros.

Los parámetros «a» pueden ser variables de cualquier tipo o bien asteriscos. En este último caso se habilita la posibilidad de que, cuando la función haya calculado el valor, devuelva el control a un sitio distinto de donde se realizó la llamada.

El uso de funciones es análogo al caso de Pascal, esto es, incluyendo el nombre de la función con sus argumentos entre paréntesis en una expresión y respetando siempre la concordancia de los tipos de los operadores.



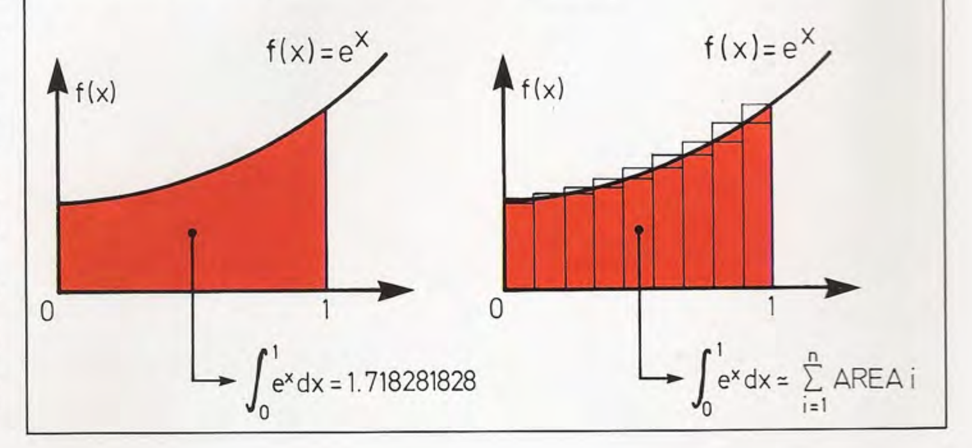
Las instrucciones del tipo DO para el control de bucles pueden anidarse, aunque no deben interferirse. El cuadro muestra las estructuras válidas y erróneas de este tipo de instrucción.

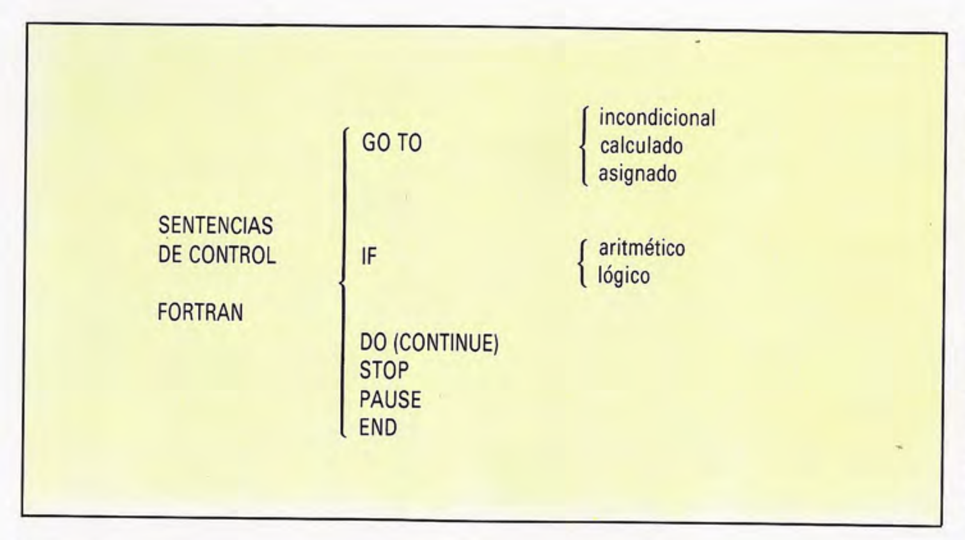
Para marcar el final de la ejecución de una FUNCTION o SUBROUTINE se introduce una sentencia «RETURN» o «RETURN n» en el lugar apropiado. Si se utiliza la segunda forma, «n» debe ser la posición en la lista de argumentos de uno de ellos de tipo asterisco. Junto al texto aparece una función FORTRAN

¿Qué es el cálculo numérico?

Cuando las necesidades de cálculo sobrepasan la simple resolución de funciones conocidas como sumas, productos, cálculos trigonométricos, etc., y lo que interesa es resolver integrales, ecuaciones en derivadas parciales o sistemas de ecuaciones diferenciales, es preciso recurrir a técnicas de cálculo numérico. Estas, a base de sencillas operaciones elementales como las citadas en primer lugar, permiten resolver cálculos complejos. El precio a pagar será una cierta inexactitud en el resultado, lo cual se verá compensado por la facilidad con que se llegará al mismo.

Por ejemplo, si se desea calcular el área encerrada bajo una curva, habría que resolver la integral correspondiente y calcular su valor entre las abscisas de interés. La alternativa que ofrece el cálculo numérico consiste en calcular el resultado a partir de sumas de áreas elementales, como se observa en la figura. La precisión del resultado dependerá de la «elementalidad» de dichas áreas.





EXPRESIONES LOGICAS EN FORTRAN

- Constante lógica
  - .TRUE.
  - .FALSE.
- Expr. con operador de relación
  - a.EQ.b (igual)
  - a.NE.b (distinto)
  - a.LT.b (menor que)
  - a.LE.b (menor o igual que)
  - a.GT.b (mayor que)
  - a.GE.b (mayor o igual que)
- Expr. con operador lógico
  - .NOT.a
  - a.OR.b
  - a.AND.b

PALABRAS CLAVES DEL FORTRAN

ASSIGN TO	ENTRY	REAL
BACKSPACE	EQUIVALENCE	RETURN
BLOCK DATA	EXTERNAL	REWIND
CALL	FORMAT	SAVE
CHARACTER	FUNCTION	STOP
CLOSE	GO TO	SUBROUTINE
COMMON	IF THEN	WRITE
COMPLEX	IMPLICIT	
CONTINUE	INQUIRE	
DATA	INTEGER	
DIMENSION	INTRINSIC	
DO	LOGICAL	
DOUBLE PRECISION	OPEN	
ELSE	PARAMETER	
ELSE IF THEN	PAUSE	
END	PRINT	
END IF	PROGRAM	
ENDFILE	READ	

adecuada para calcular el factorial de un número. Como se observa, la palabra END marca el final físico de la subrutina. Una llamada a la función así declarada puede tomar el siguiente aspecto:

```
I=FACT($10,5)
```

Se comprueba que el asterisco de la declaración se corresponde en la llamada con «$10», en donde 10 será el número de alguna sentencia ejecutable. En nuestro caso, si como consecuencia de la ejecución del IF aritmético se llega a «RETURN 1», se retornará al punto de llamada y de ahí a la sentencia número 10. Si el final de la función lo marca un sencillo RETURN, el control es devuelto de la forma habitual ya conocida en otros lenguajes como el BASIC. En cualquiera de los casos la variable FACT tomará el valor 120.

La segunda y última forma de realizar un subprograma es a través de una SUBROUTINE. Su declaración toma el siguiente aspecto:

```
SUBROUTINE nombre (al,...,an)
```

donde «nombre» es el nombre de la subrutina y a1,...,an los argumentos, los cuales son todos pasados por referencia, como ocurre con los parámetros VAR en Pascal. Una llamada a subrutina se realiza a través de la sentencia CALL como sigue:

```
CALL nombre
```

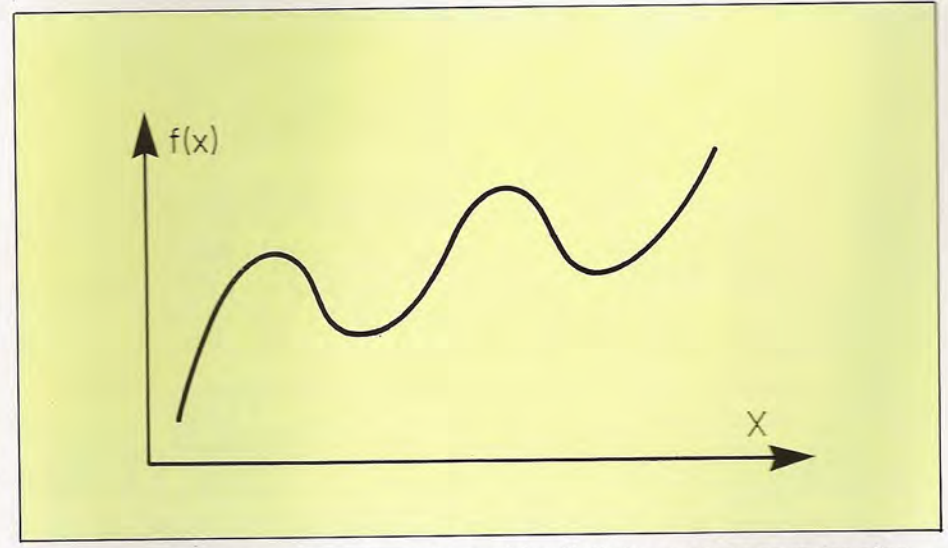
Cabe destacar que el nombre de una subrutina sirve tan sólo como referencia a la misma y que, por tanto, no lleva asociado tipo alguno, ya que no albergará ningún valor. En la figura adjunta se ha resuelto el problema del cálculo del factorial a través de una subrutina. Las mismas consideraciones ya realizadas en las FUNCTIONs sobre los argumentos de tipo asterisco son aplicables aquí.

Por último hay que mencionar la posibilidad de seleccionar el punto de una subrutina por el que se desea que comience su ejecuciónal ser invocada por medio de una instrucción ENTRY. Este mecanismo proporciona una mayor flexibilidad a las subrutinas, haciendo posible su utilización por personas con distintos propósitos.

Lisp

Creando programas inteligentes

Lisp puede ser considerado como el primer lenguaje de la Inteligencia Artificial. Históricamente, el único antecesor del que tenemos referencia es el IPL, desarrollado por Newell, Shaw y Simon en la universidad de Carnegie-Mellon en los años 50. Lisp fue inventado por John McCarthy a principios de los 60, como un medio de representación de funciones matemáticas, y su nombre viene del acrónimo formado por las letras iniciales de «LISt Processing» (procesamiento de listas).



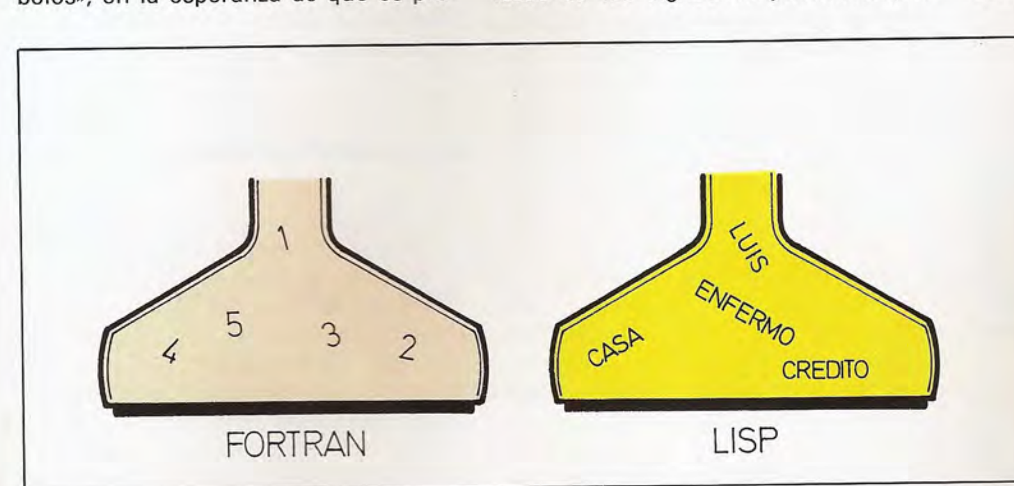
Los orígenes del lenguaje inventado por John McCarthy a principios de los años 60 están relacionados con las investigaciones en torno a la representación de funciones matemáticas.

La aparición de Lisp supuso un cambio radical en la concepción de lo que hasta ese momento era un lenguaje de programación. El Fortran y Cobol, los más populares por las fechas que interesan, ponían todo el énfasis en el tratamiento matemático de los datos. La unidad fundamental de cálculo era el número, y toda la estructura del lenguaje estaba orientada a ella, una orientación mucho más marcada en el Fortran que en el Cobol.

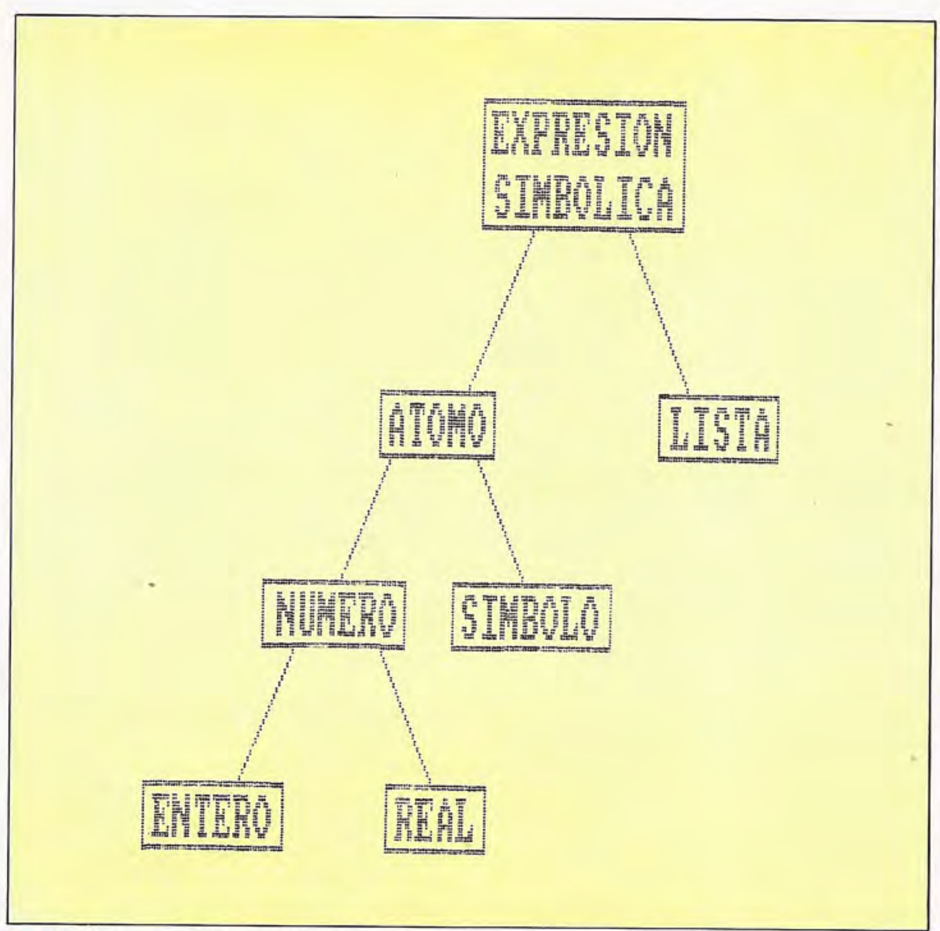
Es un hecho patente la dificultad de encontrar un comportamiento inteligente en un programa escrito en alguno de estos dos lenguajes. Por supuesto, alguien puede pensar que multiplicar dos matrices de 1000×1000 elementos complejos en menos de cinco minutos requiere algún grado de inteligencia, pero para la mayoría de la gente esto no es así. La principal idea que se introdujo con Lisp era la «manipulación de símbolos», en la esperanza de que es precisamente esta manipulación y no la manipulación de números la que debe proporcionar programas en los que sea posible vislumbrar un comportamiento inteligente que vaya más allá de realizar alocadamente miles de operaciones matemáticas por segundo.

La sintaxis del lenguaje Lisp es extremadamente sencilla, si bien alguien que lo conozca tan sólo de pasada pueda pensar que los miles de paréntesis que es posible encontrar en un programa de dimensiones normales echan al traste esta afirmación. En realidad, frente a otros lenguajes para los que son necesarias un buen montón de páginas describiendo sus palabras clave o las sutilezas de los signos de puntuación, la sintaxis de Lisp se reduce a los «átomos» y las «listas».

Un átomo es una agrupación de letras, números o signos de puntuación como «-» ó «.». Cualquier palabra de un lenguaje natural puede, por tanto, ser considerada en Lisp un átomo. Las listas son agrupaciones de átomos, las cuales a su vez pueden combinarse para formar estructuras de mayor nivel (listas de listas); es precisamente aquí donde reside una de las mayores fuerzas del lenguaje. El trabajo fundamental de Lisp consiste en la manipulación de los átomos y de las listas, los cuales reciben conjuntamente el nombre de «expresiones simbólicas». De esto último se pueden sacar nuevas conclusiones para en-



Frente a lenguajes como FORTRAN, en los que el elemento básico de cálculo o cómputo es el número, en el lenguaje Lisp se hace especial hincapié en el manejo de símbolos, si bien cabe la posibilidad de realizar potentes cálculos matemáticos.



En la figura están representados los elementos fundamentales del lenguaje Lisp. Como se puede comprobar, cualquier «cosa» que se encuentre en un programa es una expresión simbólica, y como tal será tratada.

tender mejor lo que significa la «manipulación de símbolos».

Primeras funciones

Lisp es un lenguaje que se encuadra entre los denominados «funcionales», lo que pone de manifiesto la importancia que tiene el concepto de función matemática en el sentido de generar un resultado a partir de una serie de argumentos.

Si a la vista del cursor de un intérprete de Lisp tecleamos la siguiente expresión:

```
(PLUS 3.14 2.71)
```

nos será devuelto de forma inmediata:

```
5.85
```

Obsérvese que la expresión original está encerrada entre paréntesis. Esto significa que se trata de una lista, la cual esta formada por tres elementos: «PLUS», «3.14» y «2.71». Normalmente, el Lisp siempre interpretará que el primer elemento de una lista como la anterior es el nombre de una función, la cual ha de ser aplicada a los siguientes elementos de la lista. En nuestro caso, se trata de la función PLUS que ha de ser aplicada sobre los argumentos 3.14 y 2.71. Como ya habrán adivinado, PLUS es la función que realiza las sumas, con lo que el resultado es el arriba mostrado. La forma de actuar del intérprete de Lisp será siempre la misma, es decir, nada más teclear una expresión como la anterior intentará buscar una función cuyo nombre coincida con el primer elemento de la lista, y considerará al resto de la misma como los argumentos sobre los que se aplica la citada función.

Un ejemplo más complicado puede ser el siguiente:

```
(DIFFERENCE (TIMES 3 4) 8)
```

Como en el ejemplo anterior, se le ha tecleado al intérprete una lista de tres elementos: «DIFFERENCE», «(TIMES 3 4)» y «8». A diferencia con aquél, el segundo elemento es una lista en sí, el resultado de cuya evaluación será el primero de los argumentos para la función «DIFFERENCE», siendo el segundo de ellos el «8». Para evaluar «(TIMES 3 4)» se sigue la regla general, es decir, se supone que el primer elemento —«TIMES»— es el nombre de una función (la de multiplicar) que ha de ser aplicada sobre «3» y «4», dando como resultado «12». Después de esta primera evaluación, los argumentos de la función «DIFFERENCE» son «12» y «8», escribiéndose por pantalla el correspondiente resultado: «4».

Lisp posee un completo juego de funciones matemáticas del estilo de las comentadas hasta ahora, si bien el nombre de las mismas varía entre implementaciones concretas del lenguaje. Así, por ejemplo, no es difícil encontrar que DIFFERENCE se escribe simplemente con el signo «—», o ver PLUS sustituido por «+». Otras funciones matemáticas son las encargadas de calcular cocientes enteros, resto de divisiones, raíces cuadradas, funciones trascendentes, calcular el máximo o mínimo de entre los argumentos, etc., tomando como parámetros tanto cantidades enteras como reales.

Manipulando símbolos

Lo que hasta ahora se ha comentado de Lisp apenas difiere substancialmente de lo que se encuentra en otros lenguajes convencionales. En concreto, la notación empleada es muy parecida a la del Forth. La fuerza del Lisp se centra en que puede manipular con igual agilidad cadenas de caracteres, lo que en el argot del lenguaje se denominan «símbolos».

Ejemplos de símbolos pueden ser GRAN, INFORMATICA o DE. Lo más interesante de los símbolos es agruparlos para formar listas. Así,·

(GRAN ENCICLOPEDIA DE LA INFORMATICA)

es una lista formada por cinco símbolos. Las operaciones elementales que Lisp proporciona para el manejo de listas se centran en la división de las mismas para obtener el primer elemento de ellas u obtener la lista original excepto el primer elemento. La función CAR devuelve el primer elemento de una lista, por lo que la expresión:

(CAR '(GRAN ENCICLOPEDIA DE LA INFORMATICA))

devolverá GRAN. Análogamente, la función encargada de devolver la lista original a excepción del primer elemento se denomina CDR; la expresión:

(CDR '(GRAN ENCICLOPEDIA DE LA INFORMATICA))

devolverá (ENCICLOPEDIA DE LA INFORMATICA).

Con estos dos ejemplos se ha contemplado la aparición en escena de un nuevo elemento, la comilla simple(«'»). Lo primero que hará el intérprete de Lisp con una expresión que contenga comillas simples, será reemplazarlas por el átomo QUOTE. De esta forma, los dos ejemplos anteriores quedarían así:

(CAR (QUOTE (GRAN ENCICLOPEDIA DE LA INFORMATICA))) y
(CDR (QUOTE (GRAN ENCICLOPEDIA DE LA INFORMATICA))).

```
(DEFUN FIBONACCI (N)
   (COND ((ZEROP N) 1)
         ((EQUAL N 1) 1)
         (T (PLUS (FIBONACCI (DIFFERENCE N 1))
                  (FIBONACCI (DIFFERENCE N 2))))))
```

```
(DEFUN FIBONACCI (N) (COND ((ZEROP N) 1) ((EQUAL N 1)
1) (T (PLUS (FIBONACCI (DIFFERENCE N 1)) (FIBONACCI
(DIFFERENCE N 2))))))
```



El Lisp es un lenguaje al que tradicionalmente se le ha achacado una gran dificultad por lo retorcido de su sintaxis. Si bien esto es hasta cierto punto verdadero, también el propio programador puede poner algo de su parte para aliviar situaciones como la que se representa en la parte inferior.

Analicemos más detenidamente cómo se evalúa la primera de las expresiones. Una vez que el intérprete de Lisp ha traducido las comillas simples por QUOTEs, aparece una lista formada por dos elementos, CAR y (QUOTE (GRAN ENCICLOPEDIA DE LA INFORMATICA)). CAR espera como argumento una lista, la cual ha de ser el resultado de evaluar el segundo argumento. Al evaluar el segundo argumento, se encuentra de nuevo con una lista de dos elementos, QUOTE y (GRAN ENCICLOPEDIA DE LA INFORMATICA). Al evaluarla aparece QUOTE, que es una función que simplemente devuelve su argumento sin tocarlo para nada. Por lo tanto, el resultado de evaluar

(QUOTE (GRAN ENCICLOPEDIA DE LA INFORMATICA))

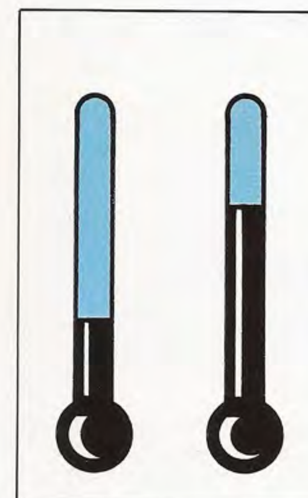
es sencillamente

(GRAN ENCICLOPEDIA DE LA INFORMATICA)





Un símbolo en Lisp es cualquier sucesión de caracteres —tanto letras como números—, entre los que pueden existir algunos signos de puntuación (puntos y guiones fundamentalmente) cuya misión es mejorar la legibilidad en el caso de que el nombre resultante sea excesivamente largo.



```
( DEFUN DE-FAHR-A-CELSIUS (TEMP)

(QUOTIENT ( DIFFERENCE TEMP 32 ) 1.8 ) )
```

En el texto se comenta el desarrollo de una función Lisp sencilla, cuya misión es la conversión de temperaturas entre dos escalas de medida.

que es precisamente la lista que se le presenta al CAR dando el resultado esperado.

Pensemos en lo que habría pasado en el caso de haber olvidado la comilla de la expresión original. Su aspecto sería como sigue:

```
(CAR (GRAN ENCICLOPEDIA DE LA INFORMATICA))
```

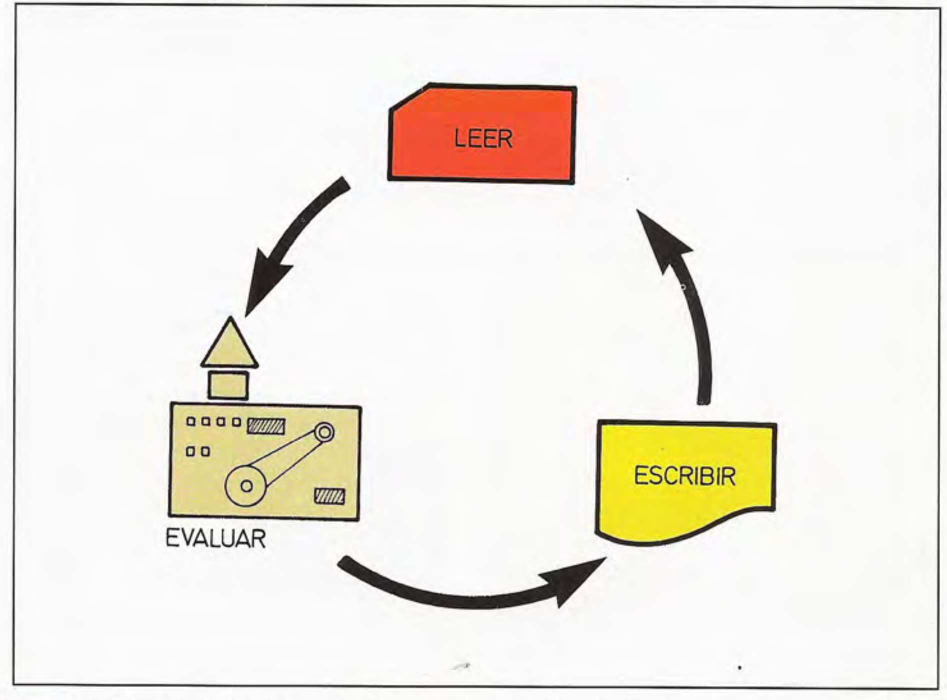
En este caso, la frenética búsqueda que lleva a cabo el intérprete de Lisp en pos de expresiones que evaluar, le habría llevado a evaluar (GRAN ENCICLOPEDIA DE LA INFORMATICA), de forma que el resultado de tal evaluación sería la lista con la que alimentar al CAR. Como consecuencia de ello, supondría la existencia de una función GRAN que al aplicarla sobre el resto de los símbolos daría la buscada lista. De inmediato surgiría un error avisando de la inexistencia de dicha función.

Lo frecuentemente que se presenta la necesidad de evitar la evaluación de expresiones simbólicas por medio del QUOTE fue la razón que introdujo la conveniencia de sustituir esta formulación por la basada en la comilla simple. Hay que recordar que lo primero que hará el intérprete de Lisp es convertir todas las ocurrencias de la citada comilla en QUOTEs, ya que de otra manera se pierde el significado de la expresión original.

Una vez vistas las funciones que se dedican a romper listas, el paso habitual es aprender a construirlas. A tal efecto existen tres funciones básicas en Lisp que son las APPEND, LIST y CONS. La primera de ella junta los elementos de todas las listas dadas como argumentos en una sola lista. LIST construye una lista cuyos elementos son cada una de las listas proporcionadas como argumentos. CONS devuelve una nueva lista cuyo primer elemento es el primer argumento y el resto el segundo de ellos. Los siguientes ejemplos ponen de manifiesto la diferencia entre las tres:

```
(APPEND '(A B) '(C D))        (A B C D)
(LIST '(A B) '(C D))          ((A B) (C D))
(CONS '(A B) '(C D))          ((A B) C D)
```

El repertorio de manejo de listas se completa con LENGTH, la cual devuelve la longitud de una lista, REVERSE, que devuelve una nueva lista con sus elementos al revés, SUBST, cuya misión es realizar la sustitución de un elemen-



Las funciones del intérprete Lisp se reducen a tres tareas fundamentales: leer una expresión simbólica por el teclado, proceder a su evaluación y escribir el resultado fruto de la citada evaluación.

to concreto en una lista por otro y LAST, que devuelve el último elemento de una lista. Los siguientes ejemplos aclaran algo más su utilización:

(LENGTH '(A B C)) ⟶ 3
(REVERSE '(A B C)) ⟶ (C B A)
(SUBST 'A 'B '(A B C)) ⟶ (A A C)
(LAST '(A B C)) ⟶ (C)

Variables globales

Hasta ahora todos los argumentos que se han pasado a las funciones vistas son los valores en sí, pero también es posible pasar como argumentos variables cuyos valores hayan sido previamente asignados. Con el fin de crear variables globales y asignarles valores existen fundamentalmente dos funciones en Lisp, aunque en la práctica es sólo una de ellas la utilizada.

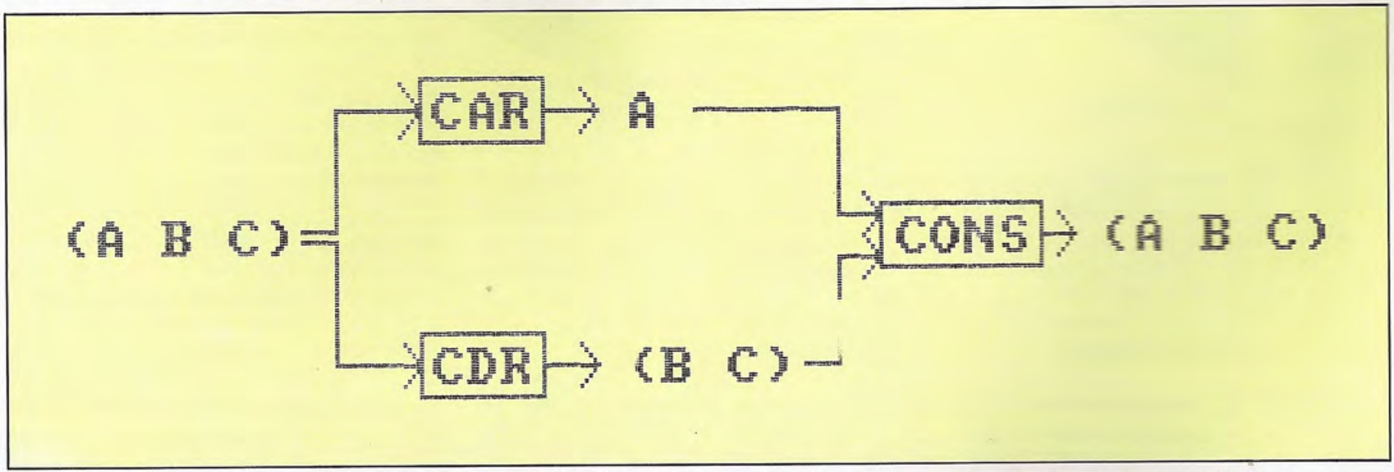
La función SET asigna un valor a un átomo simbólico. Por ejemplo:

(SET 'UNA-LISTA '(X Y Z))

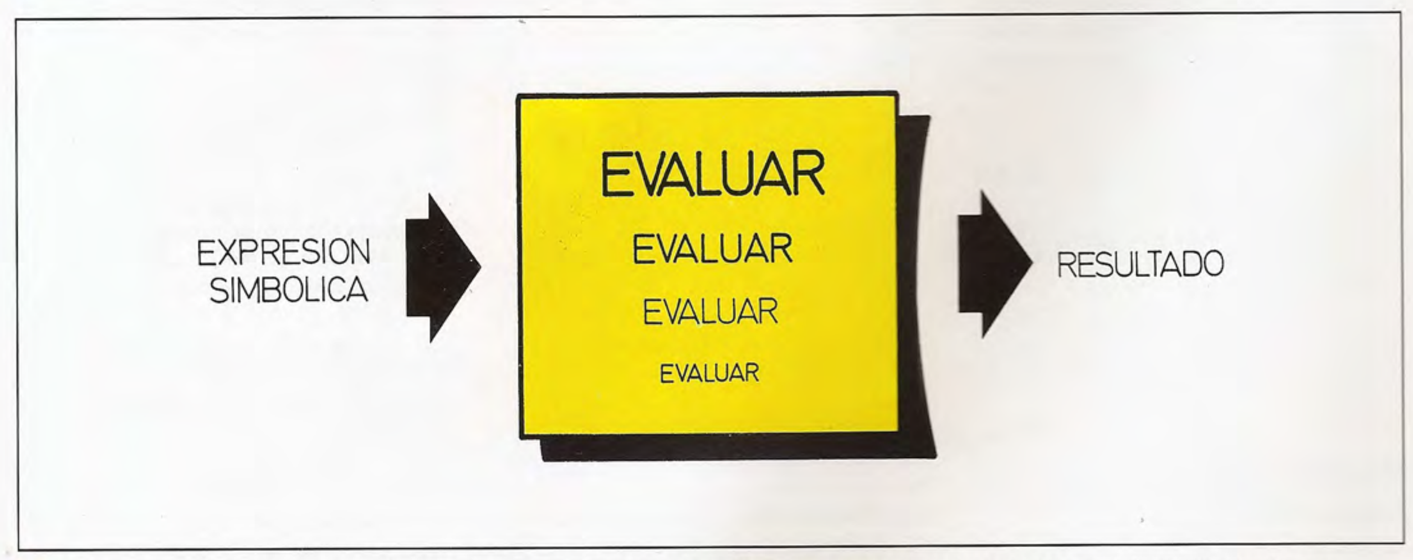
crea la variable global de nombre «UNA-LISTA» y le asigna el valor (X Y Z). La obcecación de Lisp en devolver siempre un valor como resultado de la evaluación de una expresión simbólica, hará que, en el caso precedente, se obtenga como resultado (X Y Z), aunque este efecto sea despreciado la mayoría de las veces, quedando sólo en la mente el hecho de que se dispone de una variable con un cierto valor. A partir de este momento puede ser utilizada con todas las funciones ya vistas. Por ejemplo, (CDR UNA-LISTA) devolverá (Y Z).

La otra función que, con iguales fines, se utiliza en Lisp es la SETQ. El mismo resultado anterior se obtendría haciendo:

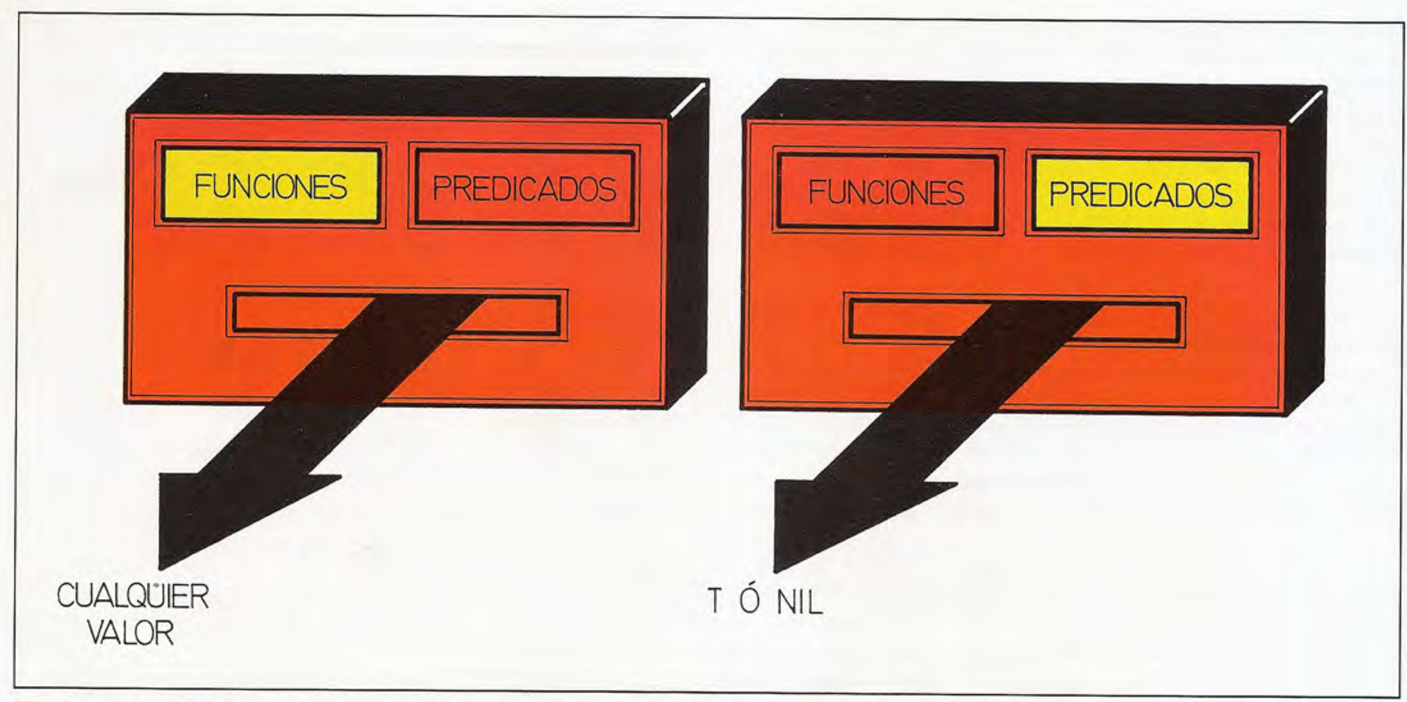
(SETQ UNA-LISTA '(X Y Z))



La lista es el elemento fundamental que es manejado en el lenguaje Lisp. Hay dos operaciones básicas cuya misión es la descomposición de listas: CAR y CDR. Este repertorio elemental se completa con CONS, cuya misión es la recontrucción de las listas seccionadas por medio de las dos funciones citadas anteriormente.



El verbo «evaluar» es una constante en cualquier descripción del lenguaje Lisp. Un intérprete para este lenguaje puede ser concebido como un «evaluador glotón de expresiones simbólicas» que produce resultados al introducirle las citadas expresiones.



En Lisp se hace distinción entre «funciones» y «predicados». Las primeras pueden devolver cualquier expresión simbólica, mientras que los posibles resultados fruto de la evaluación de los segundos se reducen a los átomos T o NIL.

Obsérvese que se ha suprimido la comilla simple que precede al átomo UNA-LISTA. Esto se debe a que SET realiza una primera evaluación del primer argumento, cosa que no hace SETQ. SET es mucho más popular que SETQ entre los programadores de Lisp. Además es posible crear y asignar distintos valores a distintas variables en una sola expresión, según se muestra a continuación:

```
(SETQ UNO 1 DOS 2 TRES 3)
```

Esta expresión crea y asigna valores a tres átomos, devolviendo como efecto secundario el valor 3, resultado de la última asignación.

Independientemente de que se use una u otra función, en el caso de que el átomo al que se le asigna un valor ya exista, sólo será éste el que resulte modificado.

Funciones creadas por el programador

Todas las funciones comentadas hasta el momento vienen programadas directamente en el intérprete de Lisp del que dispongamos. La potencia del Lisp viene determinada en gran medida por la posibilidad de construir nuevas funciones a partir de las ya existentes. Las expresiones simbólicas cuya misión sea el definir nuevas funciones han de comenzar necesariamente por el átomo DEFUN, y su estructura general es como sigue:

```
(DEFUN <nombre de la función>
        (<param. 1> <param. 2> ... <param. n>)
        <descripción del proceso>)
```

DEFUN no evalúa ninguno de los elementos que siguen en la lista; simplemente constituye una llamada de atención al intérprete para que tome nota de una nueva situación y guarde su definición en el sitio oportuno. Como siempre, la evaluación por parte del intérprete de una expresión simbólica que comienza por DEFUN dará como resultado un valor, que en este caso será el nombre de la función, aunque esto carece prácticamente de cualquier importancia. La lista que sigue al citado nombre son los parámetros a partir de los cuales se elaborará un resultado. Físicamente se trata de una lista de símbolos que pueden aparecer en la parte de descripción del proceso.

Como primer ejemplo de definición de funciones, consideremos la siguiente expresión simbólica que define una función cuyo propósito es pasar una temperatura en grados Fahrenheit a grados Celsius:

```
(DEFUN DE-FAHR-A-CELSIUS (TEMP)
        (QUOTIENT (DIFFERENCE TEMP 32) 1.8)
)
```

Con ella se define la función de nombre «DE-FAHR-A-CELSIUS» que tomará un argumento, el cual recibe localmente el nombre de «TEMP». El resultado de evaluar la expresión simbólica (QUOTIENT (DIFFERENCE TEMP 32) 1.8), cuya traducción a términos matemáticos más simples es:

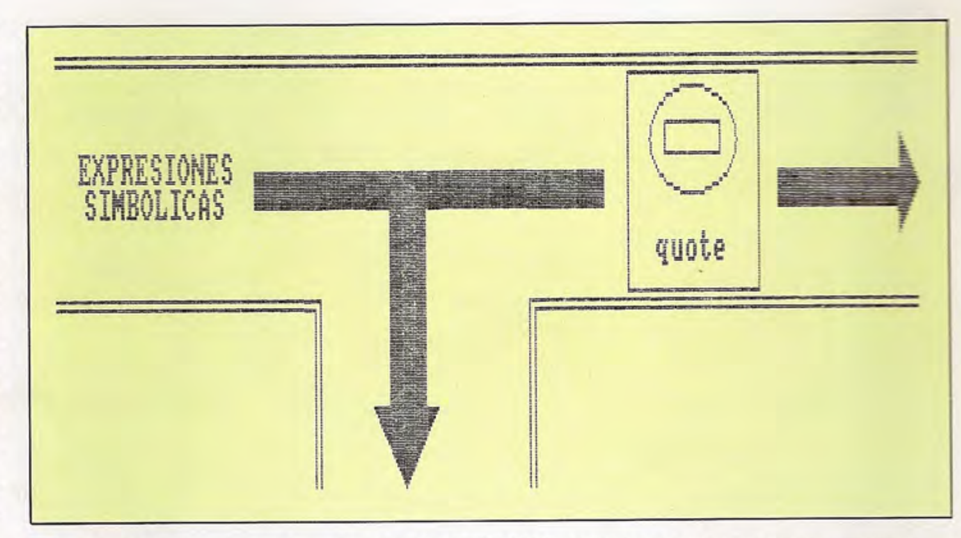
$$\frac{TEMP - 32}{1.8}$$

será el que tome la función.

Lisp es un lenguaje de programación de los denominados «de formato libre», lo cual quiere decir que es posible escribir un programa de manera que el texto realce lo más posible el propósito del mismo, de forma parecida a como se utiliza la indentación en lenguajes como Ada o C. Por esta razón, se ha colocado el último paréntesis de la expresión que define a la nueva función en una línea aparte justo debajo del que la inicia. En un programa Lisp, el número de paréntesis que se abren y se cierran puede crecer hasta límites que sobrepasan la capacidad para saber cuál se corresponde a cuál, por lo que es muy interesante el ser lo más rigurosos posibles en las normas de indentación que son utilizadas habitualmente.

A partir de este momento se puede utilizar la nueva función para obtener las conversiones deseadas. Por ejemplo, el resultado de evaluar (DE-FAHR-A-CELSIUS 104) será 40.

En la parte señalada en la forma general de una DEFUN como «descripción del proceso» pueden incluirse tantas expresiones simbólicas como sean necesarias para conseguir el resultado deseado. A título de ejemplo, la definición vista de DE-FAHR-A-CELSIUS puede sustituirse por esta otra, en la que la citada parte está formada por dos expresiones simbólicas:



Al producirse la evaluación del átomo QUOTE, el intérprete toma sus argumentos y los devuelve sin efectuar ningún procesamiento ulterior sobre ellos.

```
(DEFUN DE-FAHR-A-CELSIUS (TEMP)
        (SETQ TEMP (DIFFERENCE TEMP 32))  ;Sustracción
        (QUOTIENT TEMP 1.8)  ;División
```

Este nuevo ejemplo ha servido para introducir la forma en la que se comentan los programas en Lisp. Cualquier cosa que aparezca en una línea de programa a continuación del signo de punto y coma será ignorada por el intérprete y su propósito será el explicar los pasos que se llevan a cabo. No hay que insistir en la importancia de comentar correctamente los programas escritos en este lenguaje o en cualquier otro, pero el hecho de la documentación tiene especial importancia en un lenguaje como Lisp en el que los programas son normalmente difíciles de interpretar.

FUNCIONES

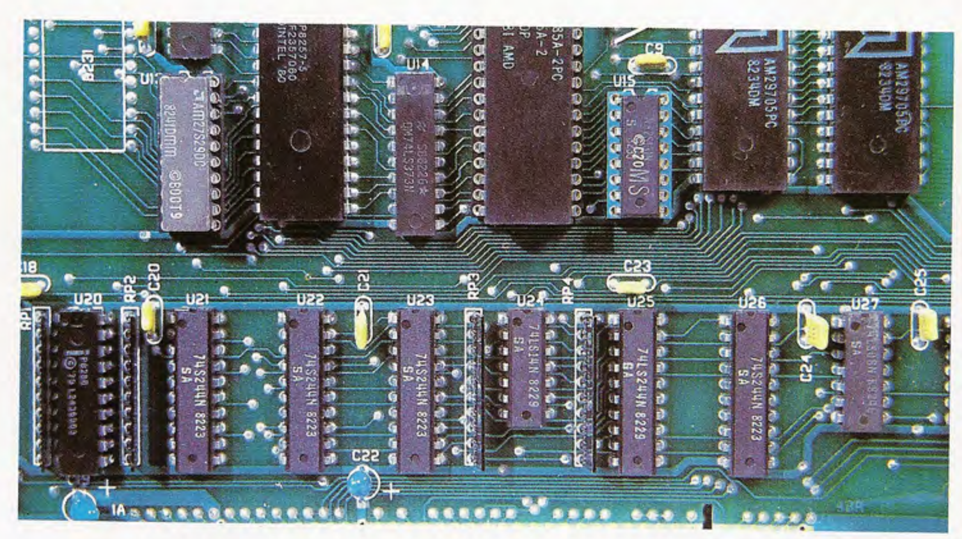
PARAMETROS

(FUNCALL FUNCION PARAMETRO (S))

La función FUNCALL permite utilizar distintas funciones sobre los mismos datos de entrada. Para ello realiza una evaluación de su primer argumento, cuyo resultado será la función concreta que hay que aplicar.

Predicados

Los predicados son un tipo particular de funciones que se caracterizan porque sólo son capaces de devolver los átomos T (True, verdadero) o NIL (falso). Un ejemplo de predicado lo constituye ATOM, que devuelve T si su argumento es un átomo o NIL en caso contrario. Así,



Las arquitecturas convencionales de ordenadores no ofrecen las características necesarias para la ejecución eficiente de programas en Lisp. Por tal razón, en el año 1974 se iniciaron diversos programas de investigación encaminados a la construcción de una «máquina Lisp», lo que en la actualidad permite elegir entre media docena aproximadamente de modelos cuyas características están orientadas a la ejecución de programas en este lenguaje, y al desarrollo de sistemas inteligentes en general.

(ATOM'UN-ATOMO) devolverá T, mientras que (ATOM '(A B C D)) devolverá NIL.

Otros predicados que normalmente se encuentran en todas las implementaciones del lenguaje son:

* EQUAL, que toma dos argumentos y devuelve T si son iguales. Por ejemplo, (EQUAL 'ATOMO 'ATOMO) devuelve T.

* NULL comprueba si su argumento es la lista vacía. (NULL '()) devolverá por tanto T, mientras que (NULL T) devolverá NIL.

* MEMBER se utiliza para comprobar si una determinada expresión simbólica es miembro de otra. En el caso de que así sea, devolverá el fragmento de la segunda a partir de la aparición del elemento buscado; si no devolverá NIL. Por ejemplo, tras haber hecho:

```
(SETQ DIGITOS '(UNO DOS TRES CUATRO CINCO))
```

la evaluación de (MEMBER 'TRES DIGITOS) devolverá:

```
(TRES CUATRO CINCO)
```

Esta forma de actuar del predicado MEMBER es reflejo de una norma general en Lisp: «Cualquier cosa distinta de NIL es T». Cuando MEMBER devuelve como resultado una lista como la mostrada en el ejemplo, se supone implícitamente que el resultado es cierto con la ventaja de que se ha generado un resultado sobre el que pueden trabajar otras funciones.

* NUMBERP determina si un átomo es numérico o no. Tras haber hecho (SETQ UNO 1), la evaluación de (NUMBERP UNO) dará T, mientras que (NUMBERP 'UNO) dará como resultado NIL.

* GREATERP y LESSP actúan sobre dos átomos numéricos dando T en el caso de que el primer argumento sea mayor o menor que el primero respectivamente.

* ZEROP y MINUSP toman un solo ar-



La implementación más famosa del lenguaje Lisp en máquinas que utilizan el sistema CP/M, como el Rainbow 100 de Digital mostrado en la figura, es la denominada mu-Lisp, aunque su sintaxis e idiosincracias dejan bastante que desear.

gumento numérico y devuelven T si, respectivamente, es nulo o negativo.

En Lisp están definidas las funciones booleanas básicas AND, OR y NOT, con un comportamiento idéntico al ya encontrado en otros lenguajes. Así, por ejemplo, AND devolverá T sólo cuando la evaluación de sus argumentos dé como resultado no NIL en todos ellos.

La función que más uso hace de los predicados es COND, además de ser una de las más fácilmente encontrables en los programas Lisp. La función COND está seguida por una serie de listas cada una de las cuales contiene un test y algo para evaluar y devolver un valor en el caso de que el resultado del test sea no NIL. Su sintaxis general toma el siguiente aspecto:

```
(COND (<test 1> ... <resultado 1>)
      (<test 2> ... <resultado 2>)
      ...
      (<test n> ... <resultado n>)
)
```

Cada una de las listas individuales recibe el nombre de cláusula. La ejecución de un COND trae como consecuencia la evaluación consecutiva de los primeros elementos de cada una de ellas hasta que se llega a uno cuyo valor es no NIL. En ese momento, el resto de la cláusula se evalúa y el resultado devuelto como ejecución del COND es el que produce la última de las evaluaciones de la citada cláusula, ignorándose el resto de las mismas. Si no se encuentra ninguna cláusula para la que el primer elemento evalúe a no NIL, se devuelve NIL.

La utilización de expresiones simbólicas con COND en el seno de DEFUNs proporcionan a éstas una flexibilidad que hasta ahora desposeían. Supongamos que queremos escribir una función que añade un nuevo elemento a una lista si no pertenece a ella. En el caso de que ya pertenezca, el valor devuelto será la lista sin modificar. He aquí una posible solución al problema:

```
(DEFUN AUMENTAR (NUEVO-ELEMENTO LISTA)
       (COND ((MEMBER NUEVO-ELEMENTO LISTA) LIS-
              TA)
             (T (CONS NUEVO-ELEMENTO LISTA))
       )
)
```

En este caso, el COND está formado por dos cláusulas. En la primera de ella, el test es (MEMBER NUEVO-ELEMENTO LISTA). Cuando NUEVO-ELEMENTO pertenece a lista, el resultado de esta evaluación será una lista cuyo primer elemento es NUEVO-ELEMENTO, un resultado distinto de NIL por lo tanto, que hará que se devuelva la evaluación del segundo elemento de la cláusula, la LISTA original. En el caso de que esto no ocurra al ser el resultado de la evaluación de «member» NIL, se intentará ejecutar la segunda de las cláusulas del COND. Aquí nos encontramos con que el primer elemento de la lista es T, cuya evaluación da T y, al ser distinto de no NIL, se devolverá el resultado de la evaluación del CONS.

Colocar un T como primer elemento en la última cláusula de una COND es una práctica habitual en Lisp. De esta forma se asegura que el resultado de un COND nunca será NIL como consecuencia de no haber encontrado ninguna cláusula cuyo primer elemento evaluara a no NIL.

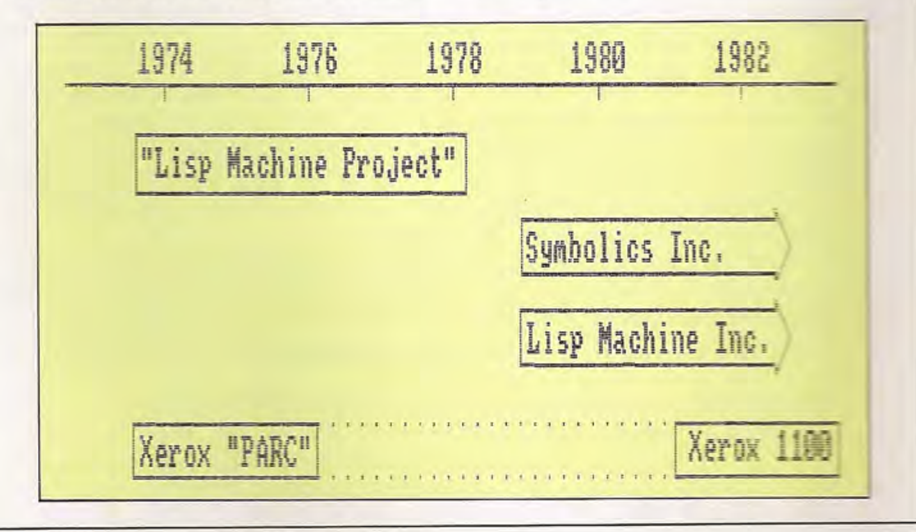
Los dialectos del Lisp

Como la gran mayoría de los lenguajes que aparecieron en el mismo momento histórico que el Lisp, la primeras implementaciones de este lenguaje se realizaron también sobre los grandes ordenadores de la época. Toda la estructura del Lisp gira en torno al concepto de lista, y es precisamente lo difícil e ineficiente que resulta en los ordenadores con arquitecturas convencionales como los citados «maniframes» o los actuales ordenadores personales cuya arquitectura estuviera de acuerdo con los principios del lenguaje, aumentando de esta forma las prestaciones de las aplicaciones desarrolladas con él.

El «Lisp machine project», cuyo objetivo era la creación de una «estación de trabajo» con las características descritas, se inició en el MIT en el año 1974, y dió como fruto dos máquinas una en el año 1976 y otra de características más avanzadas en el 1978. El grupo que se encargó del desarrollo de estos dos prototipos vió un mercado muy interesante en la comercialización de este tipo de ordenadores, por lo que en el año 1980 el citado grupo inicial se desgajó en dos compañías que en la actualidad poseen un sector bastante amplio del mercado, la Symbolics Inc. en Massachusetts y la Lisp Machine Inc., con oficinas también en Los Angeles.

De forma prácticamente paralela, investigadores del «PARC», siglas de «Palo Alto Research Center» de la compañía Xerox comenzaron a implementar el lenguaje Lisp sobre lo que es considerado por muchos el primer ordenador personal de la historia, el «Alto», aunque su éxito comercial fue nulo.

Como consecuencia de estos esfuerzos conjuntos aparecieron fundamentalmente dos dialectos del lenguaje, el «MacLisp» en torno al MIT y el «InterLisp» en los laboratorios de Xerox. En la actualidad, y dada la importancia que está adquiriendo el Lisp como lenguaje de la Inteligencia Artificial y en un esfuerzo por contrarrestar el empuje del Prolog, el Departamento de Defensa Norteamericano ha promovido la creación del «CommonLisp», en un intento por crear un estándar sólido, capaz de recoger lo mejor de cada uno de los dialectos existentes, preparando el camino para el desarrollo de aplicaciones bajo su patrocinio en temas de Inteligencia Artificial.





En el ámbito de los ordenadores nacidos bajo la estrella IBM, la implementación de Lisp más famosa es la que recibe el nombre de «Golden CommonLisp».

FUNCALL: Funciones como parámetros

Existen problemas en los que interesa aplicar distintas funciones a los datos de entrada en función de los diversos resultados que se desee obtener. Tal es el caso de las cuentas de los bancos, en los que dependiendo del cliente de que se trate se aplican distintos tipos (funciones) de interés de capital.

Según esto, se podrían tener dos funciones de interés:

```
(DEFUN INTERES-1 (BALANCE)
        (TIMES BALANCE TIPO-DE-INTERES)
)

(DEFUN INTERES-2 (BALANCE)
        (PLUS (TIMES BALANCE TIPO-DE-INTERES) EX-
        TRAS)
)
```

En las que las variables TIPO-DE-INTERES y EXTRAS están prefijadas de antemano. La función de interés INTERES-2 premia ciertos balances de cuentas con unos extras que no aparecen en INTERES-1.

Para el cálculo del interés correspondiente a una cierta cuenta cuyo balance supondremos que es de 1000, siempre se realizaría la siguiente llamada:

```
(FUNCALL FUNCION-DE-INTERES 1000)
```

Con anterioridad se habrá asignado mediante un SET o SETQ el valor INTERES-1 o INTERES-2 a la variable FUNCION-DE-INTERES. La función FUNCALL evalúa el primero de sus parámetros para obtener el nombre de una función sobre la que se aplica el resto de los elementos de la lista de la expresión de llamada.

Un poco de todo para finalizar

El propósito de este capítulo no era convertir a los lectores en grandes programadores de Lisp, sino tan sólo mostrar los principios básicos del lenguaje y esbozar su filosofía. Funciones muy potentes como las que manejan listas de asociación o las dedicadas a definir funciones con un número indeterminado de parámetros no han sido comentadas en favor de una explicación más extensa de las más humildes. Otros puntos importantes como las funciones lambda, los arrays o explicaciones más profundas sobre los mecanismos de evaluación de expresiones simbólicas han seguido el mismo camino.

Las aplicaciones más espectaculares de Lisp han surgido en torno a programas que parecen poseer un cierto grado de inteligencia. Así, Lisp ha sido el lenguaje básico para la construcción de programas capaces de resolver problemas de cálculo a nivel de bachillerato superior, detectar enfermedades infecciosas en la sangre, comprender circuitos electrónicos, determinar los lugares en los que realizar prospecciones petrolíferas o demostrar teoremas matemáticos. En general, el Lisp ha constituído, hasta la aparición del Prolog, la herramienta básica en cualquier estudio de Inteligencia Artificial. Ya se han citado campos de actuación de los sistemas expertos, pero igualmente se podrían citar investigaciones en el campo del reconocimiento del lenguaje natural escrito como hablado y en el reconocimiento de figuras y formas en dos y tres dimensiones.

Las buenas esperanzas que ofrece el Lisp y la manipulación de símbolos han hecho surgir ordenadores cuya arquitectura está orientada específicamente a la ejecución de programas escritos en este lenguaje, y en los que desde el sistema operativo hasta la última aplicación como un editor de textos están escritos en él.

Indice Temático

Lenguajes informáticos

Del código máquina a los lenguajes de alto nivel 5
Niveles de los lenguajes informáticos 5
Lenguajes de alto nivel 7

Cuadros
Los principales lenguajes de alto nivel 10

Ada

Un paso importante hacia la unificación 11
Una historia centenaria 11
La necesidad crea el lenguaje 11
Estructura de un programa en Ada 13
Los tipos en Ada 13
Estructuras de control 14
Estructuras de datos 15
Procedimientos y funciones en Ada 16
Los paquetes (packages) 17
Procesamiento en paralelo 18

Cuadros
Lenguajes imperativos y declarativos 17
La portabilidad del Ada 17
Estructuras de control del Ada 18

C (1)

Un lenguaje para la programación de sistemas 19
La historia del C 21
¿Qué es el C? 21
El camino hasta ejecutar un programa en C 22
Apariencia de un programa en C 23
Un nuevo ejemplo 24
Más sobre «printf» 26
Los tipos de datos 26
Tipos de almacenamiento 27
La estructura if-then-else 28
La estructura switch 29
Los bucles en C 30
Afianzando conceptos 31
Algo más sobre funciones: El «recurso abstracto» 31
Los punteros del C 32
Argumentos de funciones y punteros 34

Cuadros
De código fuente a código ejecutable 30
El siempre presente «;» 31
Lista de palabras clave del lenguaje C 32
Especificadores de campo para «printf» 32
Las constantes simbólicas 33
Lo cierto y lo falso en el C 34

C (y 2)

Estructuras de datos: Arrays y registros 35
Punteros y aritmética 35
La estructura array 36
Representación interna 36
Arrays de caracteres 38
Utilidad y manejo de registros 40
Cuando aparecen las restricciones 42
Estructuras autorreferenciadas 42
La biblioteca de entrada/salida 43
Las operaciones con bits 45
La escasa importancia de ser «main» 45
La sentencia «goto» 46

Cuadros
Los datos en memoria 41
Suma de los elementos de un array 42
Cuando el mundo tiene más de una dimensión 43
Strings de caracteres 44
Cómo decir mucho en pocas palabras 45
La unión hace la fuerza 46

COBOL

Un pionero de los lenguajes informáticos 47
El propósito del COBOL 47
A vueltas con la historia 48
Elementos del lenguaje 48
Estructura de un programa COBOL 49

Tipos de sentencias 52
Subprogramación 55
Recapitulación y ejemplos 56

Cuadros
Ciclo de vida de un programa 55

TABLAS
Palabras reservadas del COBOL 56

FORTH (1)

Introducción al lenguaje FORTH 57
El concepto de pila 57
Palabras 58
El diccionario 58
Números 59
Notación polaca inversa 59
Aritmética 59
Aspecto de un programa FORTH 59
Operadores aritméticos 59
Operadores evolucionados 62
El trabajo con la pila 63
Constantes y variables 64
Imprimiendo en pantalla 66
Operadores lógicos 67

Cuadros
Historia del FORTH 68

TABLAS
Tabla de órdenes FORTH (1) 66
Tabla de órdenes FORTH (2) 67

FORTH (2)

Palabras y estructuras de control 69
Cómo definir nuevas palabras 69
Cambiando definiciones de palabras 72
Realizando comparaciones 74
Elección del camino adecuado 75
Bucles elementales 75
Bucles controlados 77
Bucles anidados 78
Bucles indefinidos 79

Cuadros
Comparaciones 79

IF/THEN/ELSE 79
DO/LOOP 79

TABLAS
Tabla de órdenes FORTH 80

FORTH (3)

Entrada de datos y bases de numeración 81
Trabajando con el buffer de entrada 81
Introduciendo números 82
Más sobre palabras 82
Secuencias de caracteres que no son palabras 83
Lectura directa del teclado 84
Bases de numeración 84
Trabajando con otras bases en FORTH 85
Distintos tipos de números 88
Números enteros 88
Enteros sin signo 89
Números en punto flotante 90
Conversión de números de un tipo a otro 91

TABLAS
Tipos de números utilizables en FORTH 91
Tabla de órdenes FORTH 92

FORTH (y 4)

Números de doble precisión, arrays y últimos detalles 93
Operaciones con números de doble precisión 94
Impresión de números en pantalla 95
Taxonomía de las palabras FORTH 96
Los «arrays» en FORTH 96
Cadenas de caracteres 97
Manejo de memoria 100
Inserción de comentarios 101
Dentro de las definiciones de palabras 101
Manejo del cassette 103
Gráficos en FORTH 104
El sonido 104

TABLAS
Tabla de órdenes FORTH (1) 102
Tabla de órdenes FORTH (2) 103

FORTRAN

El precursor de los lenguajes de alto nivel 105
Un poco de historia 105
Estructura de un programa 107
Variables FORTRAN 107
Estructuras del control 109
La entrada/salida 111
Los subprogramas FUNCTIONs y SUBROUTINEs 113

Cuadros

La teoría formal de lenguajes 111
Especificadores de campo y caracteres de control 112
¿Qué es el cálculo numérico? 113
Palabras clave del FORTRAN 114

Lisp

Creando programas inteligentes 115
Primeras funciones 116
Manipulando símbolos 116
Variables globales 119
Funciones creadas por el programador 120
Predicados 121
FUNCALL: Funciones como parámetros 124
Un poco de todo para finalizar 124

Cuadros

Los dialectos del Lisp 123

